TESIS CARRERA DE MAESTRÍA EN INGENIERÍA

CLASIFICACIÓN DE SEÑALES DOPPLER DE ECOS RADAR USANDO DEEP LEARNING

Franco A. Alcaraz
Maestrando

Dr. Javier Areta
Director

Ing. Roberto Costantini
Co-director

Miembros del Jurado
Dr. Juan Pablo Pascual
Dr. Felix Rojo Lapalma
Dr. Facundo Bromberg

18 de Diciembre de 2020

INVAP

Instituto Balseiro
Universidad Nacional de Cuyo
Comisión Nacional de Energía Atómica
Argentina

Índice de contenidos

A Juliana, mi compañera de vida,
y a nuestros Ramiro y Genaro.

Agradecimientos

A **Juliana**, mi esposa, amiga y compañera, junto a nuestros hijos **Ramiro** y **Genaro**, que han brindado mucho de su tiempo y amor para que yo pueda realizar este trabajo.

A mis padres, **Raquel** y **Alberto**, que me han permitido recorrer mucho del camino previo a todo esto.

A **Darío Giussi** y **Roberto Costantini** por su orientación y apoyo, y por permitirme trabajar en esta tesis para INVAP.

A **Javier Areta**, mi director, por su tiempo, consejos, críticas, por brindarme parte de su conocimiento.

A los **miembros del jurado** por dedicar tiempo a la lectura de la tesis y por los valiosos comentarios y observaciones que realizaron.

A todas esas, casi incontables, personas que con sus palabras y obras, me han motivado a lo largo de la vida.

Resumen

En esta tesis se muestra el desarrollo de un clasificador de blancos terrestres, tales como *personas*, *animales*, *automóviles* y *tanques*, a partir de sus firmas *micro-Doppler* obtenidas con un radar pulsado que opera en banda X. El proceso de clasificación se divide en dos etapas principales. La primera transforma las señales Doppler (dominio temporal), obtenidas por el radar a partir de los ecos de los blancos, a una secuencia de imágenes que se construyen a partir del *espectrograma* de porciones de dicha señal. La segunda etapa, el clasificador propiamente dicho, se implementa utilizando *redes neuronales convolucionales (CNN)*, enmarcadas en la categoría de modelos de *Deep Learning* y que son ampliamente utilizadas en la clasificación de distintos tipos de imágenes. Diversas arquitecturas y parametrizaciones han sido analizadas para evaluar su desempeño al utilizar un dataset de señales de radar reales, verificándose finalmente, que el proceso de clasificación desarrollado es adecuado para esta aplicación, presentando un muy buen desempeño, y que es viable una implementación de tiempo real dentro de una plataforma radar.

Si bien el enfoque principal del desarrollo es implementar el proceso de clasificación usando imágenes de espectrogramas, también se analizan diversas alternativas, como por ejemplo *escalogramas* construidos usando *transformaciones wavelets discretas*. Se implementa también un proceso de detección de segmentos útiles de la señal Doppler a los fines de mejorar la calidad de las muestras de entrenamiento y evaluación, como así también mejorar la calidad del *producto clasificación* entregado al usuario radar (pos-procesamiento).

Palabras clave: DOPPLER, MICRO-DOPPLER, CLASIFICACIÓN, RADAR, REDES NEURONALES CONVOLUCIONALES, RNC, APRENDIZAJE AUTOMÁTICO, APRENDIZAJE PROFUNDO

Abstract

This thesis shows the development of a classifier of terrestrial radar targets, such as *people*, *animals*, *cars* and *tanks*, using their *micro-Doppler* signatures obtained with a pulsed radar operating in X band. The classification process is divided into two main stages. The first one transforms the Doppler signals (in time domain), obtained by the radar from the echoes of the targets, into a sequence of images that are built from the *spectrogram* of portions of that signal. In the second stage, the classifier is implemented using *convolutional neural networks (CNN)*, framed in the category of *Deep Learning* models, widely used in the classification of different types of images. Several architectures and parameterizations have been analyzed to evaluate their performance when using a real radar signals dataset. Finally, performance of the developed classification process for this problem is verified. Moreover, feasibility of real-time implementation as part of a radar system is confirmed.

While the main approach of development is to implement the classification process using spectrogram images, various alternatives are also analyzed, such as *scalograms* constructed using *discrete wavelet transformations*. A process for detecting useful segments of the Doppler signal is also implemented in order to improve the quality of training and evaluation samples, as well as improving the quality of the *classification product* delivered to the radar user (post-processing).

Keywords: DOPPLER, MICRO-DOPPLER, CLASSIFICATION, RADAR, CONVOLUTIONAL NEURAL NETWORKS, CNN, MACHINE LEARNING, DEEP LEARNING

Notación

Esta sección provee una referencia general de la notación usada en este documento. La notación es la utilizada en [1].

Números y Arreglos

a	Un escalar (entero o real)
$\boldsymbol{a}$	Un vector
$\boldsymbol{A}$	Una matriz
A	Un tensor
$\boldsymbol{I}_n$	Matriz Identidad con n filas y n columnas
$\boldsymbol{I}$	Matriz Identidad con su dimensión implícita por contexto
$\boldsymbol{e}^{(i)}$	Vector de una base canónica $[0, \ldots, 0, 1, 0, \ldots, 0]$ con un 1 en la posición i
$\text{diag}(\boldsymbol{a})$	Una matriz cuadrada y diagonal con valores en los elementos de la diagonal dado por $\boldsymbol{a}$
a	Una variable aleatoria escalar
$\mathbf{a}$	Una variable aleatoria vectorial
$\mathbf{A}$	Una variable aleatoria matricial

Conjuntos y Grafos

$\mathbb{A}$	Un conjunto
$\mathbb{R}$	El conjunto de los números reales
$\{0, 1\}$	Conjunto formado por 0 y 1
$\{0, 1, \ldots, n\}$	Conjunto de todos los enteros entre 0 y n
$[a, b]$	El intervalo real incluyendo a a y b
$(a, b]$	El intervalo real excluyendo a a pero incluyendo a b
$\mathbb{A} \backslash \mathbb{B}$	Substración de conjunto, es decir, el conjunto conteniendo los elementos $\mathbb{A}$ que no están en $\mathbb{B}$
$\mathcal{G}$	Un grafo
$Pa_{\mathcal{G}}(\mathrm{x}_i)$	Los padres de x_i en $\mathcal{G}$

Indexado

a_i	Elemento i del vector $\boldsymbol{a}$, con el indexado comenzando en 1
a_{-i}	Todos los elementos del $\boldsymbol{a}$ a excepción del elemento i
$A_{i,j}$	Elemento i, j de la matriz $\boldsymbol{A}$
$\boldsymbol{A}_{i,:}$	Fila i de la matriz $\boldsymbol{A}$
$\boldsymbol{A}_{:,i}$	Columna i de la matriz $\boldsymbol{A}$
$\mathsf{A}_{i,j,k}$	Elemento (i, j, k) del tensor 3-D $\mathbf{A}$
$\mathbf{A}_{:,:,i}$	la porción 2-D del tensor 3-D
a_i	Elemento i del vector aleatorio $\mathbf{a}$

Operaciones de Álgebra Lineal

$\boldsymbol{A}^\top$	Transpuesta de la matriz $\boldsymbol{A}$
$\boldsymbol{A}^+$	Pseudo-inversa Moore-Penrose de la matriz $\boldsymbol{A}$
$\boldsymbol{A} \odot \boldsymbol{B}$	Producto por elementos (Hadamard) de $\boldsymbol{A}$ y $\boldsymbol{B}$
$\boldsymbol{a} * \boldsymbol{b}$	Convolución entre $\boldsymbol{a}$ y $\boldsymbol{b}$
$\det(\boldsymbol{A})$	Determinante de $\boldsymbol{A}$

Cálculo

$\frac{dy}{dx}$	Derivada de y con respecto a x
$\frac{\partial y}{\partial x}$	Derivada parcial de y con respecto a x
$\nabla_{\boldsymbol{x}} y$	Gradiente de y con respecto a $\boldsymbol{x}$
$\nabla_{\boldsymbol{X}} y$	Matriz de derivadas de y con respecto $\boldsymbol{X}$
$\nabla_{\mathbf{X}} y$	Tensor que contiene las derivadas de y con respecto a $\mathbf{X}$
$\frac{\partial f}{\partial \boldsymbol{x}}$	matriz del Jacobiano $\boldsymbol{J} \in \mathbb{R}^{m \times n}$ de $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$
$\nabla^2_{\boldsymbol{x}} f(\boldsymbol{x})$ or $\boldsymbol{H}(f)(\boldsymbol{x})$	La matriz Hessiana de f en el punto de entrada $\boldsymbol{x}$
$\int f(\boldsymbol{x}) d\boldsymbol{x}$	Integral definida sobre el todo el dominio de $\boldsymbol{x}$
$\int_{\mathbb{S}} f(\boldsymbol{x}) d\boldsymbol{x}$	Integral definida con respecto de $\boldsymbol{x}$ sobre el conjunto $\mathbb{S}$

Probabilidad y Teoría de la Información

$\mathrm{a} \perp \mathrm{b}$	Las variables aleatorias a y b son independientes
$\mathrm{a} \perp \mathrm{b} \mid \mathrm{c}$	Las variables son condicionalmente independientes dado c
$P(\mathrm{a})$	Una distribución de probabilidad sobre una variable discreta
$p(\mathrm{a})$	Una distribución de probabilidad sobre una variable continua, o sobre una variable cuyo tipo no ha sido especificado
$\mathrm{a} \sim P$	Variable aleatoria a que tiene una distribución P
$\mathbb{E}_{\mathrm{x} \sim P}[f(x)]$ or $\mathbb{E}f(x)$	Esperanza de $f(x)$ con respecto a $P(\mathrm{x})$
$\overline{x}$	Valor medio de x
$\mathrm{Var}(f(x))$	Varianza de $f(x)$ bajo $P(\mathrm{x})$
$\mathrm{Cov}(f(x), g(x))$	Covarianza de $f(x)$ y $g(x)$ bajo $P(\mathrm{x})$
$H(\mathrm{x})$	Entropía de Shannon de una variable aleatoria x
$D_{\mathrm{KL}}(P \Vert Q)$	Divergencia de Kullback-Leibler de P y Q
$\mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$	Distribución Gaussiana sobre $\boldsymbol{x}$ con media $\boldsymbol{\mu}$ y covarianza $\boldsymbol{\Sigma}$

Funciones

$f : \mathbb{A} \to \mathbb{B}$	Función f con dominio $\mathbb{A}$ y rango $\mathbb{B}$
$f \circ g$	Composición de las funciones f y g
$f(\boldsymbol{x}; \boldsymbol{\theta})$	Una función de $\boldsymbol{x}$ parametrizada por $\boldsymbol{\theta}$. (Algunas veces se escribe $f(\boldsymbol{x})$ y se omite el argumento $\boldsymbol{\theta}$ para simplificar la notación)
$\equiv$	Equivalente, equivale a
$\triangleq$	Igual a, por definición
$\log x$	Logaritmo natural de x
$\sigma(x)$	Función logística (sigmoide), $\dfrac{1}{1+\exp(-x)}$
$\zeta(x)$	Softplus, $\log(1+\exp(x))$
$\|\|\boldsymbol{x}\|\|_p$	Norma L^p de $\boldsymbol{x}$
$\|\|\boldsymbol{x}\|\|$	Norma L^2 de $\boldsymbol{x}$
x^+	Parte positiva de x, es decir, máx$(0, x)$
$\mathbf{1}_{\text{condition}}$	es 1 si la condición es verdadera, 0 de otra manera

Algunas veces usamos una función f cuyo argumento es un escalar pero se aplica a un vector, matriz, o tensor: $f(\boldsymbol{x})$, $f(\boldsymbol{X})$, o $f(\mathbf{X})$. Esto denota la aplicación de f a cada uno de los elementos del arreglo. Por ejemplo, si $\mathbf{C} = \sigma(\mathbf{X})$, entonces $C_{i,j,k} = \sigma(X_{i,j,k})$ para todos los valores válidos de i, j y k.

Datasets y Distribuciones

p_{data}	La distribución que genera los datos
$\hat{p}_{\text{data}}$	La distribución empírica definida por el conjunto de entrenamiento (training set)
$\mathbb{X}$	Un conjunto de muestras de entrenamiento
$\boldsymbol{x}^{(i)}$	La i-ésima muestra (entrada) de un dataset (conjunto)
$y^{(i)}$ or $\boldsymbol{y}^{(i)}$	La clase (target) asociada con $\boldsymbol{x}^{(i)}$ para aprendizaje supervisado
$\boldsymbol{X}$	La matriz $m \times n$ con la muestra de entrada $\boldsymbol{x}^{(i)}$ en la fila $\boldsymbol{X}_{i,:}$

Capítulo 1

Introducción

Desde la creación de los primeros radares[1], cuyos fines eran la detección remota de objetos estáticos o móviles, y la medición de la distancia a la que se encontraban del radar, se han diversificado las aplicaciones que hacen uso de un radar como sensor. Entre estas aplicaciones se pueden mencionar: medición de velocidad, meteorología, relevamiento superficial, clasificación e identificación de blancos, control de tránsito aéreo, navegación, entre otras.

El trabajo que se presenta en esta tesis se focaliza en la aplicación de **clasificación de blancos terrestres**, que son de interés para las tareas de búsqueda y vigilancia, [2]. Una vez que el radar ha realizado una detección, obteniendo la ubicación (2D o 3D) del blanco y parámetros adicionales como su velocidad y Radar Cross-Section (Sección Eficaz Radar) (RCS); determinadas aplicaciones necesitan, además, de una clasificación del blanco detectado. La clasificación del blanco consiste en asignar las probabilidades de pertenecer a distintas clases predefinidas en función de las características que se puedan extraer de la información presente en uno o más ecos recibidos por el radar. Generalmente, la característica que se utiliza para realizar la clasificación de blancos es la **firma Doppler** del mismo. Dicha firma se obtiene gracias a que el blanco de interés se encuentra en movimiento y/o tiene partes constitutivas que están en movimiento, alterándose por ello las componentes frecuenciales del eco que retorna al radar de una manera particular que permite clasificar dicho blanco. Cabe mencionar que existen otros métodos de clasificación que se basan principalmente en las características morfológicas de los blancos, como ser: High Range Resolution (Alta Resolución en Rango) (HRR), Synthetic Aperture Radar (Radar de Apertura Sintética) (SAR) e Inverse SAR (SAR Inverso) (ISAR), que necesitan de radares de alta resolución espacial. [3–5]

El abordaje usual que se tomaba hasta hace un tiempo para lograr realizar una clasificación, consistía en estudiar los distintos tipos de blancos, obtener las firmas

[1] El término **radar** surge del acrónimo ***ra****dio* ***d****etection* ***a****nd* ***r****anging*.

Doppler en distintas circunstancias y ambientes, generar un modelo para cada clase y luego crear clasificadores que hacían uso de estos modelos para definir un algoritmo de clasificación. Los tipos de blancos más estudiados son: personas (caminando, corriendo, arrastrándose), helicópteros, aviones y vehículos terrestres (autos, camiones, tanques), puesto que eran los de mayor interés en el ámbito militar [6]. De esa manera se conformaron los primeros sistemas radar capaces de realizar las tareas de Radar Automatic Target Recognition (Reconocimiento Automático de Blanco Radar) (ATR) basados principalmente en modelos estadísticos complejos y algoritmos de procesamiento digital de señales, [7–15]. Posteriormente se introdujeron algoritmos tradicionales de Machine Learning (Aprendizaje Automático) (ML) logrando mejorar algunos aspectos de la clasificación, como la generalización, precisión, reducción de la complejidad en los modelos y algoritmos, a costa de necesitar de una cantidad grande de datos de entrenamiento para conseguir valores de desempeño aceptables. [16–21]

Gracias al incremento en la capacidad de cómputo, desarrollo y publicación de nuevos algoritmos de Artificial Intelligence (Inteligencia Artificial) (AI), especialmente de Deep Learning (Aprendizaje Profundo) (DL), publicaciones de librerías de código abierto, y excelentes resultados obtenidos en el campo de la AI para solucionar distintas problemáticas; se ha demostrado la efectividad de este enfoque sin la necesidad de desarrollar modelos complejos. Las tareas de clasificación, especialmente de imágenes, han obtenido resultados más allá de las expectativas, superando en muchos casos el desempeño de humanos calificados, [22–24]. En conclusión, todos estos avances y herramientas disponibles, han motivado el uso de estos algoritmos y técnicas en una diversidad muy grande de problemáticas; es por ello la motivación de este trabajo, que busca implementar un clasificador de blancos radar, utilizando técnicas de DL, especialmente Convolutional Neural Network (Redes Neuronales Convolucionales) (CNN), con las que se han obtenido excelentes resultados en la clasificación de imágenes de la vida cotidiana. Referencias adicionales a trabajos relevantes: [25–36]

En resumen, el trabajo que se presenta en este documento detalla el desarrollo de un clasificador de blancos terrestres a partir de su firma micro-Doppler obtenida a partir de ecos de un sensor radar mono-estático pulsado, utilizando DL, particularmente redes CNN. El abordaje del desarrollo prioriza la implementación del clasificador en plataforma de radares comerciales, tal que permita la clasificación en tiempo real de uno o más blancos simultáneamente. Para esto, se propone realizar la transformación de las señales Doppler (temporales) de cada blanco a una secuencias de imágenes, que por defecto, se construyen a partir del espectrograma de fragmentos de dicha señal.

1.1. Motivación

Este trabajo de Maestría surge como parte de un proceso de capacitación dentro de la empresa INVAP S.E., para el cual se eligió al Instituto Balseiro como institución educativa debido a su excelente nivel académico y vinculación estratégica con INVAP. Se consideró importante el inicio de la incorporación de conocimiento en los temas de AI, especialmente ML y DL, aplicados a las distintas áreas de negocio que tiene la empresa, principalmente al área de desarrollo de radares.

Este trabajo busca, entonces, desarrollar un prototipo de **Clasificador de Blancos Radar** a ser embebido en los distintos radares desarrollados y a desarrollar por la empresa, considerándose como una característica importante en los futuros productos de la empresa. Asimismo, se busca abrir una línea de trabajo dentro del ámbito académico para profundizar el conocimiento en AI aplicada, en el área de telecomunicaciones (radares principalmente) y de visión artificial, creando un vínculo estratégico entre la academia y la industria.

1.2. Objetivos

Los objetivos fijados previamente al desarrollo fueron los siguientes:

- Implementar un clasificador de blancos radar.
- Implementar el clasificador utilizando técnicas y algoritmos de Deep Learning.
- Clasificador que pueda implementarse en el hardware de procesamiento disponible en los radares, con uso bajo a moderado de los recursos de dicho hardware.
- Conseguir un desempeño en precisión de al menos 80 % para todas las clases, o comparable con el obtenido con otras técnicas, para el mismo dataset.
- Entrenar y validar el clasificador utilizando un dataset con datos reales de un radar.
- Latencia[2] para la clasificación de un blanco menor o igual a 4 segundos.
- Tiempo de clasificación menor o igual a 100 ms (capacidad de clasificar como mínimo 10 blancos diferentes por segundo).

[2]Se entiende por *latencia* al intervalo de tiempo que transcurre entre la detección del blanco y su clasificación.

1.3. Organización y alcance de la tesis

El documento basa su estructura en el flujo de procesamiento de los datos, dividiéndose en tres partes:

- Señales y Datos. Esta parte incluye el entendimiento de la naturaleza de las señales que conforman las muestras del dataset, que posteriormente se utiliza para entrenar, testear y validar el clasificador. También se detalla el pre-procesamiento que se realiza sobre las muestras crudas y los procesos que se utilizaron para conformar finalmente el dataset.

- Clasificador. Esta parte incluye una introducción teórica del tipo de modelo de clasificador elegido, en este caso CNN; los modelos propuestos, el detalle del entrenamiento de los mismos, y los resultados obtenidos.

- Conclusiones. Como parte final del documento, se pone a disposición las conclusiones obtenidas del proceso de desarrollo del clasificador, como así también de los resultados obtenidos. También se esboza un listado de futuros posibles trabajos, como continuación del trabajo presentado en este documento.

Parte I

Señales y Datos

El primer paso en el desarrollo del clasificador es la conformación del dataset a partir del cual se entrenará y evaluará el mismo. El dataset, entonces, debe ser representativo de los datos con que el usuario final alimentará el clasificador para obtener los resultados de interés. El dataset puede crearse a partir de datos obtenidos con uno o más radares (datos reales), datos obtenidos a partir de modelos de blancos (datos sintéticos), o combinación de ambos.

La ventaja de utilizar datos sintéticos es que se puede generar una gran cantidad de muestras que representen una buena diversidad de propiedades de los blancos y de los escenarios (esencial para mejorar la capacidad de generalización del clasificador, es decir, ser capaz de brindar inferencias correctas inclusive para aquellas situaciones a las que nunca fue expuesto). La desventaja, en este caso, radica en que el desempeño del clasificador, ante señales de blancos reales, se encontrará sujeta a qué tan representativos son los modelos de los blancos reales en los escenarios en que estén inmersos. Ante la utilización de datos sintéticos para el entrenamiento de los modelos, es recomendable en última instancia realizar la evaluación del clasificador utilizando datos reales.

La utilización de datos reales presenta la ventaja de trabajar con todas las particularidades que puedan presentar los blancos y los escenarios de interés, las cuales son difíciles y costosas de modelar cuando se quieren generar datos sintéticos. La dificultad de tener un entrenamiento basado unicamente en datos reales es que, para conseguir los mismos, se debe realizar una campaña de adquisición de los datos en campo, con todos los blancos y escenarios de interés, contemplando las variantes en las propiedades de los mismos, que a priori se consideren de valor, para asegurar una buena generalidad de los datos.

El foco de este documento está puesto en el entrenamiento de modelos de aprendizaje automático utilizando datos reales. En esta parte del documento se tratará, de principio a fin, el proceso de adquisición de los datos, procesamiento y preparación de los datasets a utilizar para el entrenamiento del clasificador. El plan inicial para el desarrollo fue realizar una campaña de adquisición de datos utilizando un radar de INVAP con blancos terrestres en diversos escenarios; sin embargo, por cuestiones de disponibilidad del radar no fue posible ejecutar dicho plan. La alternativa por la que se optó continuar fue la utilización de un dataset disponible para su uso. Por lo explorado en la web y a través de las referencias encontradas en papers, existen dos datasets con adquisiciones de señales doppler: una de ellas se denomina **RadEch Database** (The Database of Radar Echoes from Various Targets), ver [37] (declarada como disponible en [38], aunque actualmente no se encuentra disponible, por lo que no pudo utilizarse para este trabajo); y la otra es **The Database of Radar Echoes from Various Targets** disponible en [39], descrita brevemente en [40], que es la que finalmente se utilizó para este desarrollo.

Es importante resaltar el hecho de que la mayoría de los trabajos consultados no desarrollan en detalle el proceso de preparación de los datos para luego alimentar el clasificador, donde la mayoría parten de un dataset ya depurado. El proceso de preparación de los datos desarrollado en esta parte del documento buscará minimizar el uso de procesamiento ad-hoc de los datos, pretendiendo que los modelos de deep-learning utilizados en el clasificador puedan simplificar esta tarea que tendría que repetirse cada vez que se incorporen nuevas adquisiciones al dataset.

Capítulo 2

Señales Doppler Radar

Este capítulo realiza una introducción al principio de funcionamiento de un radar y cómo se obtiene la señal Doppler, que finalmente será utilizada por el clasificador para asignar, al blanco iluminado por el radar, una probabilidad de pertenecer a una clase. Estudiar la naturaleza de la señal Doppler permitirá entender y ajustar el proceso de depuración y procesado de las muestras crudas de las adquisiciones radar.

2.1. Principios de funcionamiento de un Radar

Un radar basa su operación en la emisión de ondas electromagnéticas para, posteriormente, recibir los ecos de esas ondas al reflejarse en elementos distribuidos en el espacio de cobertura. La función básica de un radar, la *detección*, se logra a través del procesamiento de las señales de los ecos obtenidos, el cuál permite discernir cuales ecos pertenecen a objetos de interés (blanco[1]). Una vez detectado un blanco, se puede medir el tiempo que le toma a la onda emitida por el radar propagarse desde el radar a dicho blanco (objeto donde la onda se refleja), más el tiempo que le toma al eco retornar al radar. La distancia R a la que se encuentra el blanco, respecto del radar, se denomina *rango* [2]; ésta se puede calcular midiendo el tiempo t_d de propagación de la onda desde el radar al blanco más el tiempo de retorno de la reflexión (eco) desde el blanco al radar (rayo directo), de la siguiente manera:

$$R = \frac{c \cdot t_d}{2} \tag{2.1}$$

donde c es la velocidad de la luz.

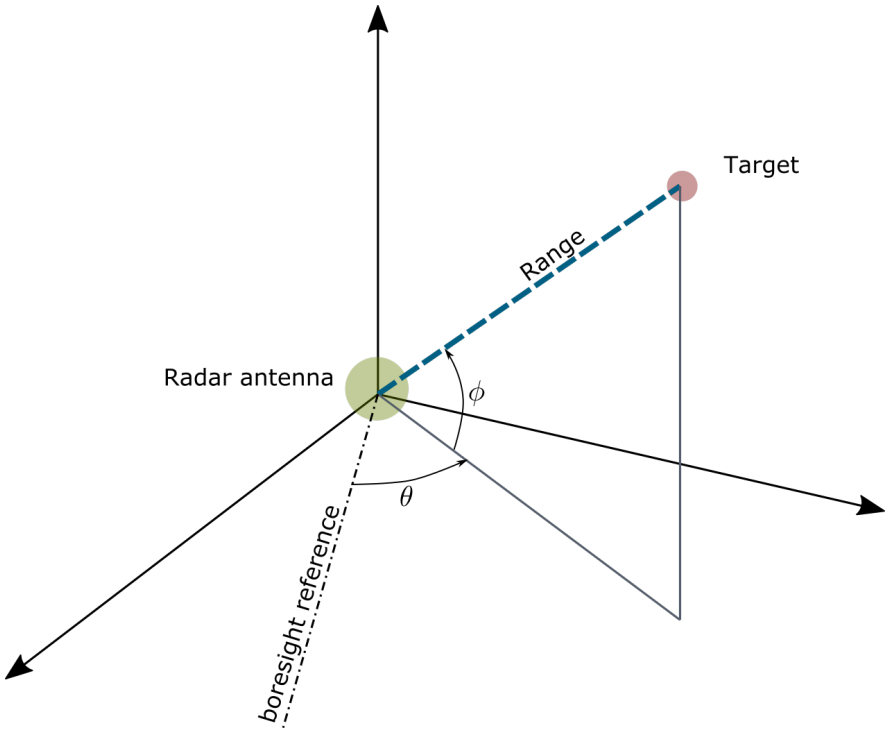


Figura 2.1: Coordenadas (esféricas) radar para la ubicación de un blanco en el caso de un radar monoestático.

[1]En la jerga de radares, un objeto detectado (o a detectar) por el radar se denomina *target*, relacionado a las aplicaciones militares de un radar, y de ahí la traducción al término *blanco* en español.

[2]En este caso se hace abuso de la traducción del término *range* del inglés, utilizado en la jerga de radares. Podría reemplazarse convenientemente este término con *distancia*.

Una vez que un objeto ha sido detectado por el radar, se pueden habilitar funcionalidades adicionales del radar como *tracking* (seguimiento) y la medición de la velocidad del blanco.

En el caso de radares monoestáticos, donde la antena que transmite las ondas y la que las recibe se encuentran en la misma ubicación, lo que el radar mide como rango es una distancia radial hacia el blanco. Esta distancia radial R nos define una superficie esférica en el espacio con todas las posibles coordenadas en la que se puede encontrar el blanco de interés. Quedan, entonces, dos coordenadas por resolver, *acimut* θ (ángulo del radial respecto a un norte de referencia) y *elevación* ϕ (ángulo de elevación respecto al plano horizontal). Estas dos coordenadas se resuelven a través del lóbulo de apuntamiento principal de la antena del radar, ver figura 2.1. La velocidad se puede medir a través de la frecuencia Doppler del eco (en este caso se trata de la componente radial de la velocidad), o bien, a través de las sucesivas detecciones del blanco.

Para refinar más los principios de funcionamiento de un radar, tenemos que adentrarnos en las dos arquitecturas más típicas: **Radar Pulsado** y **Radar de Onda Continua**.

Nota: Está fuera del alcance de este documento explicar en detalle los modos de funcionamiento de un radar, sino presentar la información relevante para entender la naturaleza de las señales Doppler que se obtendrán de los sensores radar para su posterior clasificación. Para mayor información consultar [41].

2.1.1. Radar pulsado

Un radar pulsado basa su funcionamiento en la emisión de pulsos de Radio Frecuencia(s) (RF), de duración y forma de onda predefinidas, para luego abrir una ventana de escucha para los posibles ecos de ese pulso que puedan generarse en el escenario de operación del radar.

Una de las ventajas de este modo de funcionamiento es que se puede utilizar la misma antena para la transmisión de pulsos y para la recepción de ecos. Sin embargo, este modo fuerza al radar a no poder escuchar ecos mientras está transmitiendo por lo que aparecen zonas ciegas y ambigüedades en rango. Otra de las desventajas es que, al transmitir pulsos de duración acotada (usualmente cortas, del orden de nano a micro segundos), la potencia de RF de dichos pulsos debe ser elevada; de esta manera se asegura que se emita la energía necesaria para obtener una cierta *Probabilidad de detección (P_d)* de los blancos.

En la figura 2.2 puede observarse la arquitectura básica de un radar pulsado. A grandes rasgos, se puede observar las dos cadenas principales: la de transmisión com-

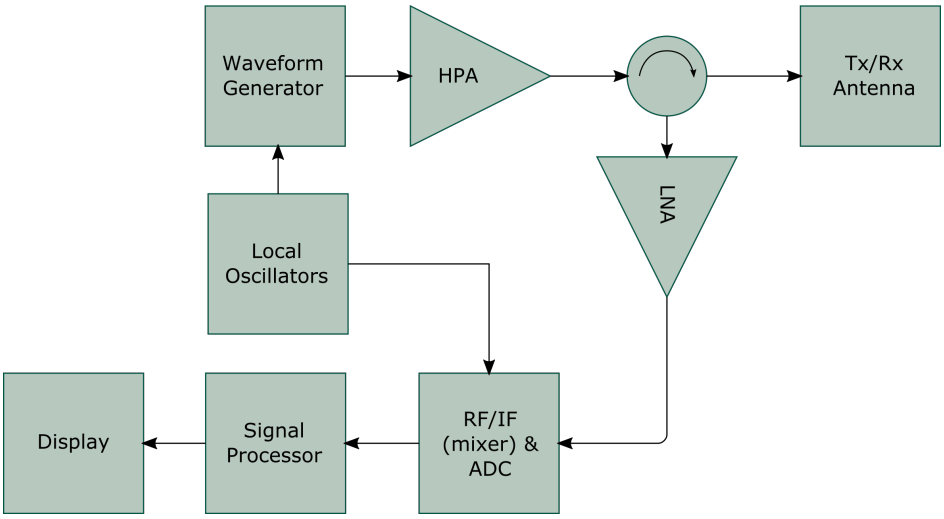


Figura 2.2: Arquitectura básica de un radar pulsado.

puesta por el *Generador de Formas de Onda*, el *High Power Amplifier (Amplificador de Potencia) (HPA)*; y la de recepción compuesta por el *Low Noise Amplifier (Amplificador de Bajo Ruido) (LNA)*, la etapa de *Conversión de RF/Intermediate Frequency (Frecuencia Intermedia) (IF)*, el *Procesador de Señales* y los componentes para la *Visualización*. Notar que hay bloques comunes a ambas cadenas, como ser el *Circulador*, la *Antena de Transmisión (Tx)/Recepción (Rx)* y el *Generador de Oscilador(es) Local(es)*. Este último bloque, al estar compartido por ambas cadenas, asegura que el sistema radar sea **coherente**; el hecho que el radar sea coherente significa que las diferencias de fase entre el pulso transmitido y el eco recibido se deben solamente (en el caso ideal) a las características y ubicación del blanco.

Podemos caracterizar un pulso transmitido por el radar como:

$$S_{Tx}(t) = A_{Tx} \cdot p(t) \cdot e^{j(2\pi f_c t + \phi_T(t))} \tag{2.2}$$

Donde:

- A_{Tx} Amplitud de la señal transmitida.
- $p(t)$ Envolvente del pulso de transmisión (modulación en amplitud).
- f_c Frecuencia de portadora.
- $\phi_T(t)$ Desviaciones de fase de la cadena de transmisión (aquí pueden considerarse la modulación propia de la forma de onda, como el ruido de fase).

Mientras que para la caracterización del eco recibido de un blanco podemos escribir:

$$S_{Rx}(t) = A_{Rx}(t) \cdot p(t - t_d) \cdot e^{j(2\pi f_c t + \phi_R(t))} \tag{2.3}$$

$$\phi_R(t) = \theta_D(t) + \theta_r + \phi_T(t - t_d) \tag{2.4}$$

Donde:

$A_{Rx}(t)$ Término que define la amplitud del eco en función de las características y distancia del blanco.

$\theta_D(t)$ Desviación de fase asociada al efecto Doppler generado por el blanco.

θ_r Variable aleatoria que modifica la fase del eco en función de las características y distancia del blanco.

t_d Tiempo de retardo del eco.

La etapa de recepción del radar será la encargada de desafectar la frecuencia de portadora, llevando la señal a *Base-Band (Banda Base) (BB)*:

$$S_{BB}(t) = A_{Rx}(t) \cdot p(t - t_d) \cdot e^{j(\theta_D(t) + \theta_r + \phi_T(t - t_d))} \tag{2.5}$$

El desarrollo de este trabajo se centrará en la utilización de la información contenida en las desviaciones de fase $\theta_D(t)$ producidas por el efecto Doppler generado por el blanco de interés, para poder realizar la clasificación. Es necesario mencionar que esta señal es compleja y se conforma por dos componentes en cuadratura (dos señales reales) que se denominan I (en fase) y Q (en cuadratura):

$$S_{BBI}(t) = G_{Rx} \frac{A_{Rx}(t) \cdot p(t - t_d)}{2} \cdot cos(\phi_R(t)) \tag{2.6}$$

$$S_{BBQ}(t) = G_{Rx} \frac{A_{Rx}(t) \cdot p(t - t_d)}{2} \cdot sin(\phi_R(t)) \tag{2.7}$$

$$S_{BB}(t) = S_{BBI}(t) + jS_{BBQ}(t) \tag{2.8}$$

Donde:

G_{Rx} Ganancia de toda la cadena de recepción.

2.1.2. Radar de onda continua

Un radar de Continous Wave (Onda Continua) (CW) basa su funcionamiento en la emisión de RF continuamente a través de una antena de transmisión, y la recepción continua de las reflexiones en los blancos a través de una antena de recepción. La señal recibida es mezclada con la señal que se está transmitiendo, por lo que el procesamiento del radar tendrá como entrada el resultado de dicha mezcla y no la señal recibida (como

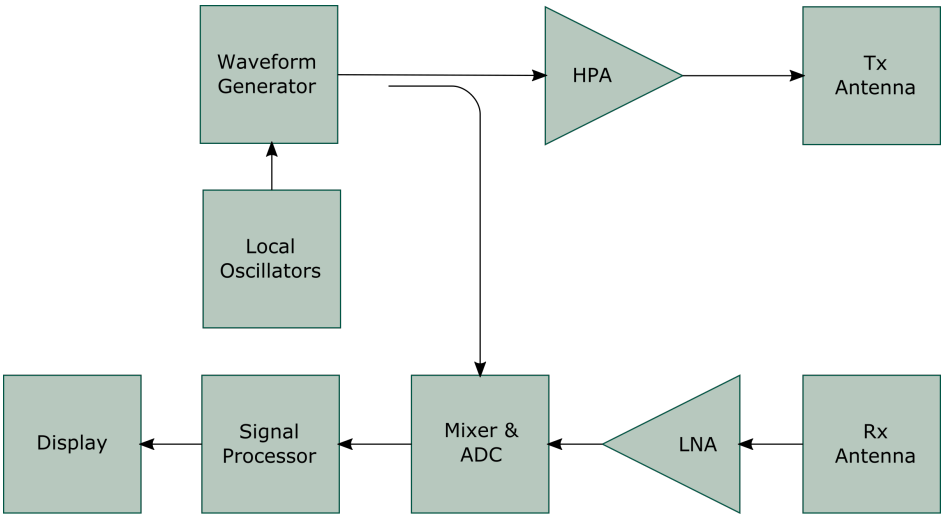


Figura 2.3: Arquitectura de un radar de onda continua.

era el caso del radar pulsado). La arquitectura de este tipo de radar puede verse en la figura 2.3. Notar que la arquitectura presenta las antenas de transmisión y recepción de manera separada, y esto se debe a que es la forma más común en que se construyen estos radares para minimizar el acoplamiento entre los caminos de transmisión y recepción.

En el caso en donde la señal que se transmite es de frecuencia continua (tono puro), la mezcla entre la señal transmitida y la recibida será distinta de cero solamente cuando los blancos generen una desviación de frecuencia al reflejar la señal del radar; esto sólo se produce cuando el blanco tiene una componente de velocidad radial distinta de cero (efecto Doppler). Los radares de CW de frecuencia constante, entonces, son incapaces de medir el rango al que se encuentra un blanco, pero sí su velocidad.

Cuando el radar de CW transmite señales que no tienen un valor constante en frecuencia, generalmente haciendo una modulación en frecuencia, se denomina Frequency Modulated Continous Wave (Onda Continua con Frecuencia Modulada) (FMCW). Dos modulaciones muy utilizadas para este tipo de radar son la *diente de sierra* y la *triangular*.

En la figura 2.4 se observa la señal transmitida por un radar FMCW que utiliza una modulación *diente de sierra*, y el eco producido por un blanco puntual que se encuentra a una distancia R del radar. Se puede escribir entonces:

$$\frac{f_b}{\Delta F} = \frac{t_d}{T_m} \tag{2.9}$$

$$t_d = \frac{2R}{c} \tag{2.10}$$

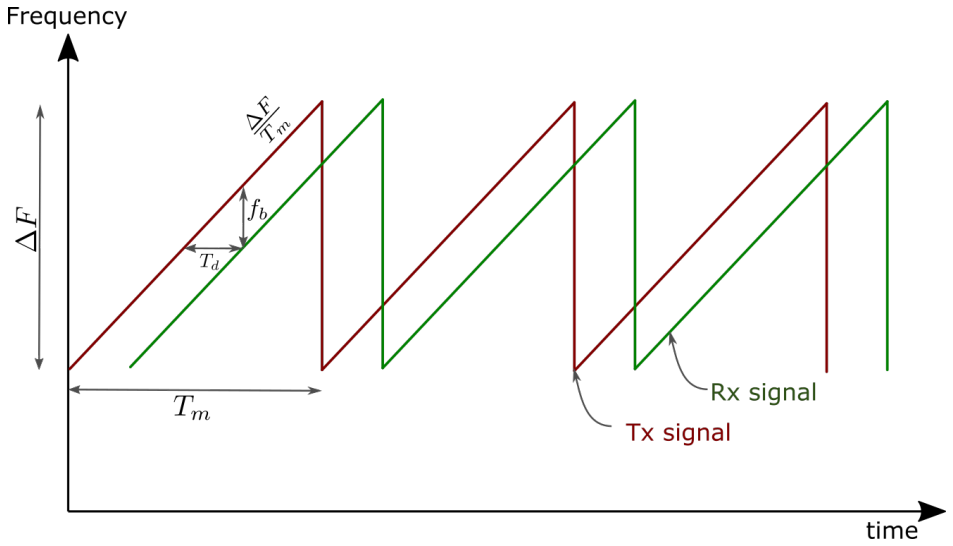


Figura 2.4: Transmisión y recepción en un radar de onda continua, usando una modulación en frecuencia del tipo diente de sierra. El gráfico muestra la variación de la frecuencia instantánea de la señal versus el tiempo.

$$f_b = \frac{\Delta F}{T_m} \cdot t_d = 2 \cdot \frac{\Delta F}{T_m} \frac{R}{c} \tag{2.11}$$

Donde:

- f_b Frecuencia de batido.
- t_d Retardo de propagación.
- ΔF Desviación de frecuencia (ancho de banda de transmisión).
- T_m Período de modulación.
- c Velocidad de propagación de la onda (velocidad de la luz).

De acuerdo a la ecuación 2.11, la señal que se digitalizará luego del batido de las señales de transmisión y recepción, tendrá una frecuencia f_b proporcional a R, por lo que los rangos de los distintos blancos se verán como tonos a distintas frecuencias.

2.2. Efecto Doppler en señales Radar

Cuando el reflector de la señal emitida por el radar se encuentra estático (respecto del radar), la señal que recibe el radar como eco tiene la misma frecuencia con que se transmitió. Cuando el reflector deja de estar estático respecto del radar, y presenta una componente de velocidad radial distinta de cero, la frecuencia de la señal recibida se verá modificada debido al efecto Doppler [42]. Bajo la consideración de que el sensor es un radar monoestático que transmite una señal a una frecuencia f_{Tx}, y que la misma

se refleja en un blanco que se mueve a una *velocidad radial* v; entonces la teoría de la relatividad especial predice que la frecuencia f_{Rx} que tendrá la señal recibida por el radar sera:

$$f_{Rx} = \left(\frac{1 + v/c}{1 - v/c}\right) f_{Tx} \tag{2.12}$$

siendo c la velocidad de propagación de la onda en el medio.

La ecuación 2.12 puede simplificarse sin perder precisión significativamente, suponiendo que la velocidad v es mucho menor que c, resultando:

$$f_{Rx} = \left(1 + \frac{2v}{c}\right) f_{Tx} \tag{2.13}$$

El cambio que sufre la frecuencia transmitida, o sea, la diferencia entre la frecuencia recibida y la frecuencia transmitida, se denomina *frecuencia Doppler* f_D. Partiendo de la ecuación 2.13, podemos calcular:

$$f_D = \frac{2v}{c} f_{Tx} = \frac{2v}{\lambda_{Tx}} \tag{2.14}$$

donde λ_{Tx} es la longitud de onda de la señal transmitida. Notar que f_D deberá ser positiva cuando el reflector se acerque al radar, por lo que v será definida positiva para ese caso.

Para ganar generalidad, debemos considerar el caso en que el reflector tenga una componente de velocidad $\boldsymbol{v}$ (vector 3D) que no necesariamente es radial al radar. En este caso, la componente que genera un cambio en la frecuencia de la señal reflejada es la radial del vector $\boldsymbol{v}$, por ello:

$$f_D = \frac{2\,||\boldsymbol{v}||\,cos\psi}{c} f_{Tx} \tag{2.15}$$

donde ψ es el ángulo formado entre el radial y el vector velocidad del blanco. Notar que si el blanco se mueve de manera ortogonal al radial, la frecuencia Doppler será cero.

Se ha mostrado cómo el movimiento del blanco genera una desviación de la frecuencia de la señal reflejada cuando la señal transmitida tiene una frecuencia constante; sin embargo, en la mayoría de los radares la modulación en frecuencia (y a veces también en amplitud) utilizada para los pulsos trasmitidos, tales como: *chirp*, *diente de sierra*, *triangular* requiere de un ancho de banda mayor, por lo que el efecto Doppler debe estudiarse para este ancho de banda y no para una frecuencia en particular [42]. El efecto Doppler que afecta a los blancos en movimiento puede ser usado, en el contexto de un radar de detección, para filtrar dichos blancos de interés de aquellos que se encuentran

estacionarios en la escena, principalmente el *clutter*.

2.2.1. Micro-Doppler

En muchos casos, un objeto (blanco) que es iluminado por un radar puede tener un movimiento oscilatorio o vibratorio respecto de su centro de masa; o bien, los componentes estructurales del mismo pueden tener movimientos relativos a dicho centro de masa, que se denominan micro-movimientos. Como ejemplos de interés podemos mencionar: un helicóptero, donde sus aspas tienen un movimiento oscilatorio distinto al del cuerpo de la aeronave; una persona, donde sus extremidades se mueven de manera similar a un péndulo con respecto al torso (este mismo ejemplo aplica a animales terrestres en general), [8, 11, 43, 44]; un avión, donde las aspas de las turbinas tienen un movimiento oscilatorio; un ave al volar, con el batido de sus alas; un automóvil con el movimiento de sus ruedas; un tanque con el movimiento de sus orugas.

Los micro-movimientos pueden inducir modulaciones (como frecuencias Doppler) de la frecuencia de portadora usada en la transmisión de las señales del radar. Según 2.15, las frecuencias e intensidades de las modulaciones son determinadas por la frecuencia de portadora, la velocidad correspondiente a cada micro-movimiento, y el ángulo entre la dirección de ese micro-movimiento y la línea de vista del radar. De esta manera, un blanco que tiene micro-movimientos inducirá tantas componentes de frecuencia Doppler como partes móviles tenga, permitiéndonos obtener las propiedades cinemáticas del objeto de interés, lo que se denominará ***Micro-Doppler Signature (Firma Micro-Doppler)***. [45, 46]

La firma micro-Doppler es una característica distintiva de cada tipo de objeto con micro-movimientos, que es observado por un radar, donde son importantes: la cantidad de componentes, intensidades, frecuencias y los patrones temporales. De esta manera, la firma micro-Doppler de un avión es distinta de la de un helicóptero, como de la de una persona. Es por ello, que esta característica es ampliamente utilizada en la clasificación de blancos radar, y es precisamente la que se utiliza en este trabajo para la clasificación de blancos terrestres. [12, 47–50]

Para obtener la señal Doppler de un blanco de interés, es necesario que las variaciones de fase de la señal sólo dependan del mismo. Es por ello necesario que el radar sea un *sistema coherente*, donde las desviaciones de fase introducidas por la cadena de transmisión están enganchadas con las desviaciones de fase introducidas por la cadena de recepción. De esta manera, la fase de la señal reflejada puede desafectarse de las desviaciones introducidas por el sistema radar y recuperarse la componente de fase correspondiente al blanco propiamente dicho.

La ecuación 2.16 muestra la relación entre la fase y la frecuencia; por lo que si

disponemos de la función que caracteriza la variación temporal de la frecuencia, se pueden obtener las desviaciones de fase que se podrán medir a partir de la señal recibida en el radar.

$$\phi(t) = \int_{-\infty}^{t} \omega(\tau)\, d\tau = \int_{-\infty}^{t} 2\pi f(\tau)\, d\tau = \phi(0) + \int_{0}^{t} 2\pi f(\tau)\, d\tau \tag{2.16}$$

donde ω es la frecuencia angular de la señal, expresada en radianes por segundo; y f la frecuencia de la señal expresada en Hertz. El término $\phi(0)$ representa la fase inicial de la señal.

Como las señales transmitidas por el radar están generadas a partir de un oscilador muy estable, pequeñas variaciones en la fase de la señal recibida pueden ser medidas. Usando las ecuaciones 2.14 y 2.16, podemos obtener la fase correspondiente a una frecuencia Doppler f_D:

$$\theta_D(t) = \int_{-\infty}^{t} 2\pi f_D(\tau)\, d\tau = 2\pi \int_{-\infty}^{t} \frac{2v_r(\tau)}{\lambda_{Tx}} d\tau = \frac{4\pi}{\lambda_{Tx}} \int_{-\infty}^{t} v_r(\tau)\, d\tau \tag{2.17}$$

donde $v_r = ||\boldsymbol{v}||\, cos\psi$ es la velocidad radial del objeto o una componente del mismo. Dicha velocidad se puede expresar como el cambio en la distancia radial en el tiempo, o sea $v_r = dr/dt$, siendo r la distancia radial del blanco. Obtenemos entonces:

$$\theta_D(r) = \frac{4\pi}{\lambda_{Tx}} \int \frac{dr}{dt}\, dt = \frac{4\pi}{\lambda_{Tx}} r + \theta_0 \tag{2.18}$$

Notar que un cambio en media longitud de onda genera un cambio de fase de 360° (2π radianes). Por ejemplo, en un radar de banda X cuya portadora es de 10 $[GHz]$; la longitud de onda es de 3 $[cm]$ aproximadamente, por lo que pueden detectarse desplazamientos relativos del orden de centímetros entre los componentes móviles de un objeto. Esto permite poder recuperar las componentes micro-Doppler de un blanco con dimensiones menores a la resolución en rango del radar, que depende del ancho de banda del pulso transmitido (modulado). Esto mismo puede expresarse diciendo que el radar será capaz (en el mejor de los casos) de extraer las componentes Doppler de todos los objetos móviles dentro de una celda de resolución del radar (resolución espacial). Entonces, si dos personas se encuentran dentro de la misma celda de resolución del radar, las firmas Doppler de ambas personas estarán mezcladas y no podremos separarlas con filtrado espacial/frecuencial.

Podremos decir, entonces, que la firma micro-Doppler de un objeto con micro-movimientos es la suma de las componentes Doppler de cada una de las partes que componen a dicho objeto, que partiendo de 2.14, se podría escribir a la señal micro-

Doppler $S_{\mu D}$:

$$S_{\mu D}(t) = e^{j\theta_{\mu D}(t)} \tag{2.19}$$

donde $\theta_{\mu D}(t)$ es la desviación Doppler correspondiente a todas las componentes del objeto, y que complementa al término θ_D de la ecuación 2.4. Esta desviación total puede escribirse como:

$$\theta_{\mu D}(t) = \sum_k \theta_{Dk}(t) = \theta_{\text{target}} + \frac{4\pi}{\lambda_{Tx}} \sum_k (\alpha_k \cdot r_k(t) + \theta_k) \tag{2.20}$$

donde $r_k(t)$ es la distancia radial en función del tiempo de la k-ésima componente del objeto de interés, α_k y θ_k modelan la amplitud y fase inicial de la señal correspondiente a dicha componente, y θ_{target} representa una fase media del blanco en su conjunto.

2.3. Procesamiento Radar - Señal Doppler

Considerando el caso de un radar pulsado, tenemos que la señal que contiene los ecos de las reflexiones de cada pulso transmitido es digitalizada durante un período de tiempo denominado *ventana de conversión (sampling window)*, a una tasa de muestreo $f_s = 1/T_s$, que generalmente es la frecuencia de muestreo de los Analog-to-Digital Conversion/Converter (Conversión/Conversor Analógico-Digital) (ADC) utilizados en cada canal de recepción. Esta frecuencia se ajusta de acuerdo al ancho de banda de la modulación utilizada (considerando que la señal es digitalizada en banda base). Esta dimensión de los datos se denomina *tiempo rápido (fast time)*. La resolución en rango del radar δ_R estará relacionada con el ancho de banda del pulso trasmitido BW_{Tx}, y por ello con f_s (suponiendo que el radar opera con una $f_s \geq BW_{Tx}$ para alcanzar esa resolución), de la siguiente manera:

$$\delta_R \geq \frac{c}{2\ BW_{Tx}} \tag{2.21}$$

por lo que cada muestra en el tiempo rápido está relacionada con una celda de resolución en rango del radar, recibiendo también el nombre de *muestra en rango (range sample)*. [42].

Cada ciclo de transmisión de un pulso y recepción de los ecos tiene un tiempo determinado que se denomina Pulse Repetition Interval (Intervalo de Repetición de Pulso) (PRI), y su inversa Pulse Repetition Frequency (Frecuencia de Repetición de Pulsos) (PRF). Este proceso agrega una nueva dimensión a los datos adquiridos por el

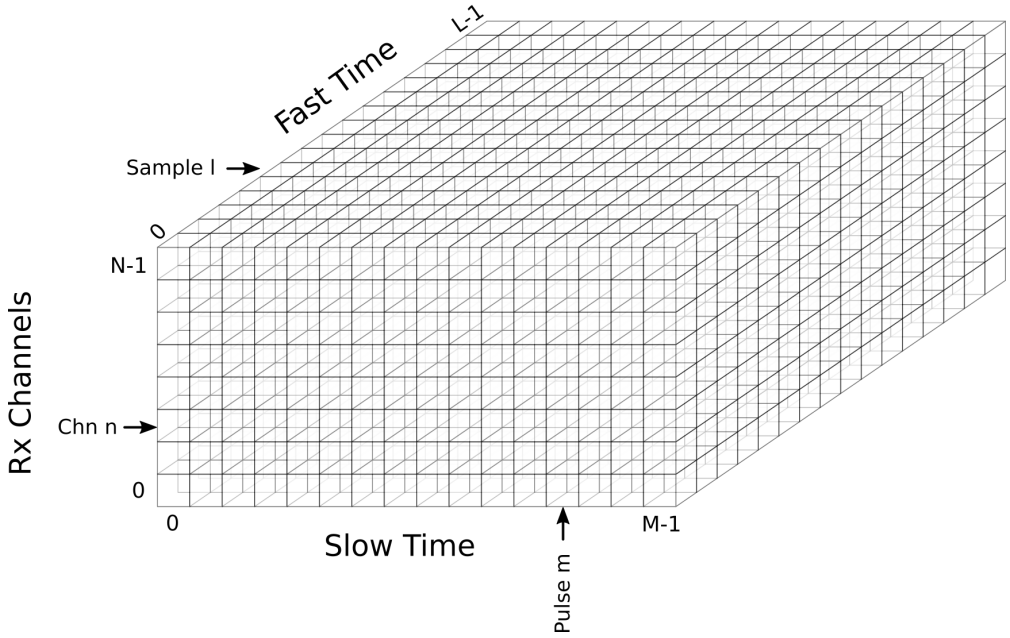


Figura 2.5: Cubo de datos Radar.

radar que se denomina *tiempo lento (slow time)*, el cual es importante para la extracción de la información Doppler de un blanco, como se explicará en breve. De esta manera, esta dimensión indexa los los ecos recibidos por cada pulso transmitido.

Una dimensión más puede aparecer si es que el radar utiliza más de un canal de recepción. Con estas tres dimensiones se conforma lo que se denomina *cubo de datos radar (radar data cube)*[51], como se muestra en la figura 2.5. Un *canal de recepción* es aquel que digitaliza de manera independiente la señal obtenida por una antena particular del radar (o sub-arreglo de antenas). Generalmente, cada canal de recepción está asociado a una antena con una ubicación espacial diferente, permitiendo al procesamiento hacer uso de esta diversidad espacial para mejorar su desempeño. Este recurso se utiliza, principalmente, cuando se quieren conformar distintos haces (*beamforming*) para realizar una exploración selectiva del espacio (en elevación y/o acimut); y también cuando se tienen distintas polarizaciones de la antena. El proceso de beamforming toma las muestras de los distintos canales de recepción (o canales de antena), para un dado instante, y por cada haz que quiera conformar, aplica desfasajes y atenuaciones específicas a cada canal de recepción y los combina (suma), como se muestra en la figura 2.6. De este modo, la dimensión de canales de recepción se transforma en la dimensión de *haces (beams)*, donde no necesariamente la cantidad de canales de recepción es igual a la cantidad de haces generados, ya que la cantidad de haces es arbitraria.

Para detectar los blancos dentro de la ventana de conversión, lo estándar es aplicar un *filtro adaptado (matched filter)* a la forma de onda del pulso transmitido, lo que devolverá picos en las muestras correspondientes a las celdas donde la probabilidad de que haya un blanco es alta, evaluación que se realiza bajo un criterio de detección y un

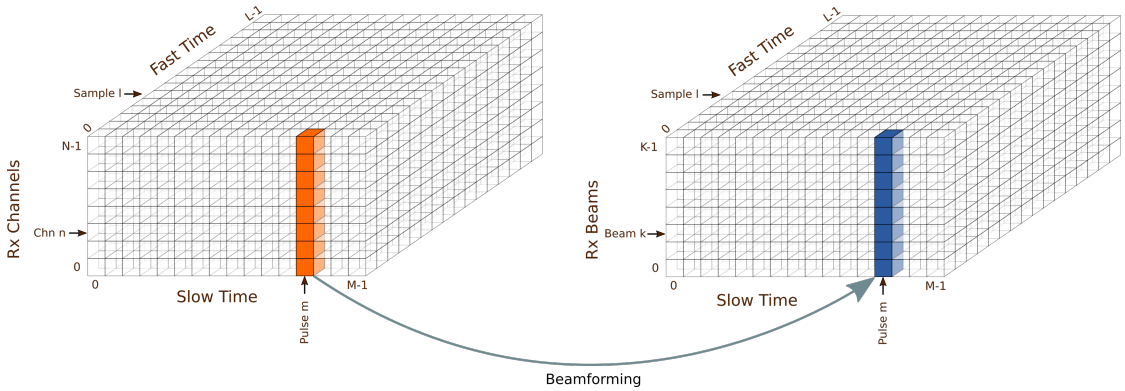


Figura 2.6: Beamforming mostrado sobre el cubo de datos radar. Se resaltan las muestras de los distintos canales de recepción correspondientes al pulso m y a la muestra #0 del tiempo rápido, que al aplicarse el beamforming se transforman en muestras de los distintos haces resultantes. El conjunto de muestras de los canales de recepción (columna naranja del cubo a la izquierda) se utiliza para la conformación de un haz (con un conjunto particular de desfasajes y atenuaciones), que se representa como un sólo bloque azul de la columna resaltada en el cubo de la derecha. Cada bloque de la columna azul se asocia con un conjunto diferente de desfasajes y atenuaciones aplicados a la columna naranja.

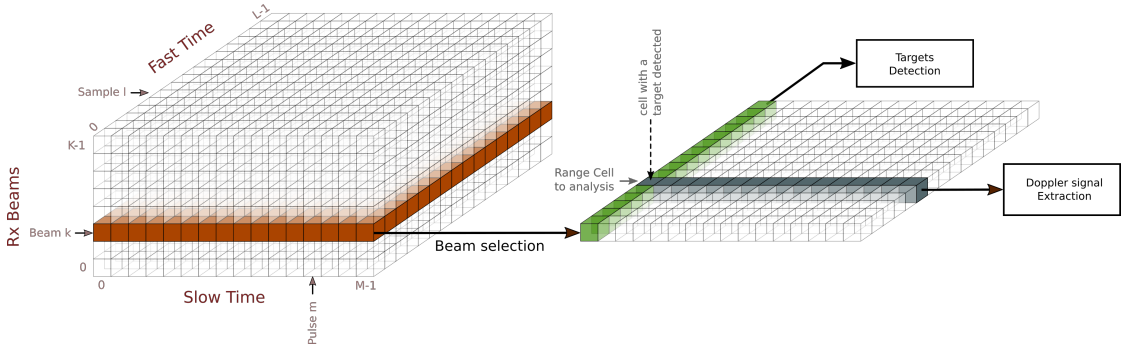


Figura 2.7: Cubo de datos Radar en el proceso de selección de un haz (selección de un plano horizontal del cubo), detección de blancos sobre muestras del haz seleccionado para un pulso determinado (muestras en verde, tiempo rápido), extracción de la señal Doppler de una celda en particular (muestras en gris, tiempo lento).

umbral ajustado en función de la *probabilidad de detección* y *falsa alarma* deseadas. Las muestras con que se alimentará la etapa de detección serán las que correspondan con la dimensión de tiempo rápido, para un determinado pulso y haz. En la figura 2.7 se muestra el caso donde se eligen las muestras en la dimensión de tiempo rápido correspondientes al haz k y el pulso 0 (muestras en verde). El ejemplo también muestra un blanco probable en la muestra l.

Si queremos obtener la firma Doppler del posible blanco en la celda l de rango, debemos extraer la señal correspondiente a todas las muestras l de los distintos pulsos, o sea, recorriendo la dimensión de tiempo lento. El filtro adaptado devolverá una componente compleja para dicha celda, que pulso a pulso, irá modificando su amplitud y fase en función de la evolución del movimiento de dicho blanco dentro de la celda de resolución del radar. Vale mencionar que el ejemplo supone que en el cubo de datos el

blanco no ha migrado a otra celda del radar a medida que se fueron recibiendo sus ecos; para tal caso es necesario ir seleccionando la celda adecuada a medida que se recorre la dimensión de tiempo lento, y para ello se utiliza un *tracker* que permite asociar blancos detectados a una trayectoria espacial. De esta manera, la PRF define la frecuencia de las muestras en el tiempo lento, por lo que equivale a la frecuencia de muestreo de la señal Doppler, y es por ello que el espectro de frecuencias Doppler (sin plegado, o dicho de otro modo, sin ambigüedades) estará comprendido en el rango de $[-PRF/2, PRF/2]$. Considerando que la frecuencia Doppler máxima será $|f_{Dmax}| = PRF/2$, a partir de 2.14, tenemos que la velocidad radial máxima no ambigua v_{rmax} será:

$$v_{rmax} = \frac{PRF \cdot c}{4 \cdot f_{Tx}} = \frac{PRF \cdot \lambda_{Tx}}{4} \tag{2.22}$$

Todas aquellas componentes que tengan una frecuencia Doppler fuera del rango $[-PRF/2, PRF/2]$, se plegarán dentro de este rango, dependiendo en la zona de Nyquist en que se encuentren (recordando que la frecuencia de Nyquist en este caso es $PRF/2$). Es decir, aparecerá un alias, o réplica, de esta componente en el rango $[-PRF/2, PRF/2]$.

Capítulo 3

Diseño

Habiendo presentado los principios de funcionamiento del radar, particularmente un radar de vigilancia, es conveniente desarrollar cómo es el flujo de señales y datos para convertir una sucesión de ecos provenientes de una celda radar a un conjunto de valores que representan las probabilidades de que el blanco en esa celda sea de una clase determinada.

En la sección 2.1 se mostraron arquitecturas genéricas de distintos tipos de radar. El proceso de convertir las señales de radiofrecuencia en productos útiles, se realiza en el *Signal Processor* de cada radar. Este subsistema implementa las distintas cadenas de procesamiento que convierten un *Cubo de Datos Radar* en uno o más productos. La clasificación de blancos radar es en sí un producto a generarse dentro del *Signal Processor*, por lo que el clasificador es un bloque dentro de la cadena de procesamiento correspondiente a ese producto. En este capítulo se hará una presentación de alto nivel de dicha cadena de procesamiento con la intención de entender el flujo de datos, interfaces y formato de datos. En la sección 2.3 se explicó cómo un radar obtiene la señal Doppler a partir del *Cubo de Datos Radar*, por lo que no se detallará dicha etapa en este capítulo.

3.1. Cadena de clasificación

Se denomina **cadena de clasificación** a la conexión en cascada de bloques de procesamiento, que toma la información necesaria del *Cubo de Datos Radar* para entregar el reporte de clasificación. Toda la cadena de clasificación se implementa dentro del bloque *Signal Processor*, que generalmente, se instancia en la *Radar Computer*, dentro de la *Electrónica Central* del Radar. El diagrama de bloques correspondiente a esta cadena se muestra en la figura 3.1.

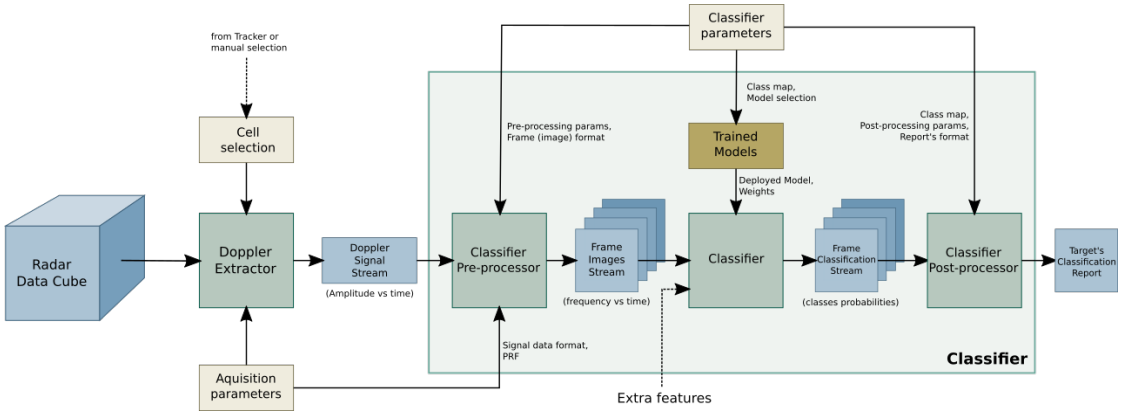


Figura 3.1: Cadena de clasificación - Arquitectura general.

La primera etapa de la cadena de procesamiento, **Doppler Extractor**, realiza la conversión de la información contenida en el *Radar Data Cube* a una señal continua que contiene la información Doppler de la celda seleccionada. Esta señal, denominada **Doppler Signal Stream**, contiene la señal temporal que representa todas las componentes Doppler de los ecos recibidos, expresada como amplitud (real o compleja). Los parámetros principales para esta etapa son la PRF, la celda en donde se encuentra el blanco, características del pulso transmitido y frecuencia de muestreo. La frecuencia de muestreo con la que se generarán los datos de la *Doppler Signal* será la PRF (salvo que se aplique alguna decimación). El formato de esta señal define, entonces, el formato de los datos de entrada al *Clasificador*.

La segunda etapa de la cadena de procesamiento, **Classifier Pre-processor**, se encarga de convertir la Doppler Signal en una secuencia de imágenes. Cada imagen se denomina **Frame** y contiene el espectrograma de un intervalo de tiempo predefinido de la Doppler Signal convertido a una imagen de dimensiones constantes. La imagen de salida se conforma, entonces, en diversas etapas: filtrando y decimando en función de las características de la adquisición, convirtiendo la porción de la señal al dominio de frecuencia vs tiempo (espectrograma) y luego convirtiendo el espectrograma a imagen de dimensiones constantes (pixeles) y tipo de dato fijo (generalmente entero de 8 bits sin signo, o UINT8). La tasa de generación de *frames* (imagénes) que se crean en esta

etapa depende del solapamiento temporal entre las porciones extraídas de la señal.

La tercera etapa de la cadena de procesamiento, **Classifier**, se encarga de clasificar cada frame. La clasificación consiste en entregar un conjunto de valores (vector), donde cada valor es la probabilidad estimada de que dicho frame pertenezca a una clase predeterminada. El conjunto de clases se determina mediante la selección del mapeo (class map). El clasificador, a alto nivel, puede tener una colección de modelos entrenados de acuerdo al universo de formatos de frame y de mapeos que se requieran; es por ello que el modelo del clasificador puede cambiarse dentro del radar dependiendo de la aplicación. Sólo puede elegirse un único modelo por cada cadena de clasificación; donde dicho modelo recibe el nombre de **Deployed Model**. Podrían incluirse *features* adicionales a la entrada del clasificador, como por ejemplo la RCS o vector velocidad/aceleración del blanco; para lo cual deberá seleccionarse el modelo entrenado con ese conjunto de features.

El clasificador entrega en su salida un stream con los resultados de la clasificación de cada frame. Este stream es consumido por la cuarta etapa de la cadena de procesamiento, **Classifier Post-processor**, que se encarga de colapsar una secuencia de predicciones en un reporte que se asocia a cada blanco; dicho reporte se denomina **Target's Classification Report**. En esta etapa pueden aplicarse técnicas de suavizado y cálculo de métricas diversas, para luego conformar el reporte de acuerdo al formato elegido. El formato del reporte puede ser simplemente la clase más probable luego de un determinado tiempo de adquisición, un ranking de clases probables o una gráfica con la evolución de las probabilidades en el tiempo.

3.2. Cadena de dataset

La cadena de conformación del dataset, o **Dataset Chain**, es la encargada de convertir el conjunto de datos crudos de las señales Doppler de las distintas adquisiciones, en un conjunto de muestras útiles, con el formato compatible con la entrada del clasificador. El *dataset* que se obtiene a la salida se utilizará como un conjunto de datos inalterable para el entrenamiento y evaluación de los distintos modelos de clasificador. Los detalles y resultados de esta cadena se presentan en el capítulo 5. En la figura 3.2 se muestran las etapas, parámetros y flujos de datos de la *Cadena de Dataset*, dichas etapas se explican a continuación.

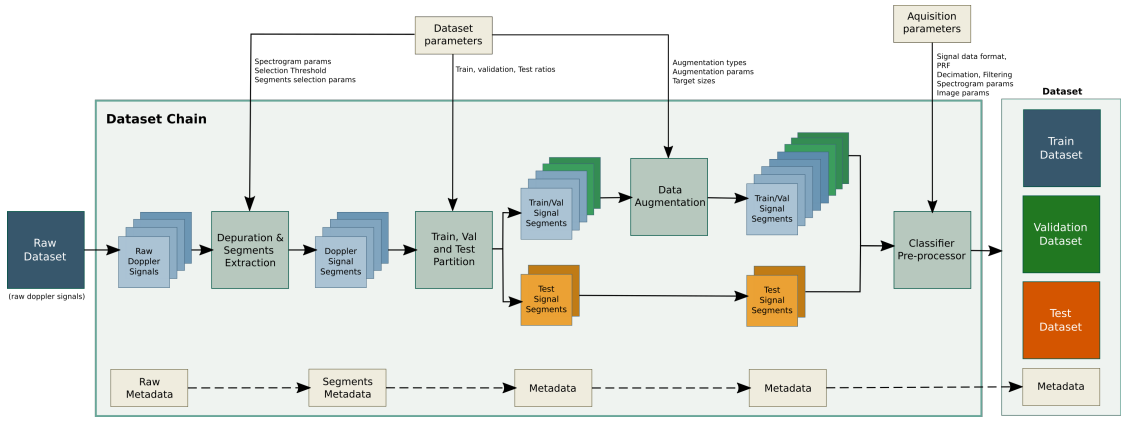


Figura 3.2: Cadena de dataset - Arquitectura general.

La primera etapa de esta cadena está destinada a la **depuración y segmentación** de las señales crudas. La *depuración* consiste en identificar todos los intervalos de tiempo en donde la señal de interés no es útil, ya sea porque se encuentra ausente, distorsionada o contaminada (interferencias). Luego se definen criterios de pasa/no-pasa para conservar/descartar dichos intervalos (en general se descartan intervalos con una baja Signal to Noise Ratio (SNR) o Signal to Interferences and Noise Ratio (SINR)), persiguiendo la obtención de un dataset que no sea problemático para entrenar los modelos, pero que sea representativo de las señales que se obtienen en la operación normal de un radar. Por otra parte, la *segmentación* consiste en crear fragmentos de señal continua en donde no hay intervalos inútiles (bajo los criterios que se mencionaron anteriormente); de esta manera, de una señal se podrá obtener uno o más segmentos de señal continua. Para promocionar un segmento como válido también se definen criterios, en donde se pueden destacar *duración mínima* y *duración de señal útil continua*. En conclusión, a la salida de esta etapa se consigue un conjunto de segmentos de señal Doppler de duraciones diversas que cumplen los criterios necesarios para considerarse útiles para el entrenamiento.

Por cada segmento válido se definen *frame*s. Cada **frame** es una porción de señal de un segmento, de longitud fija, el cual se convertirá posteriormente (luego del pre-procesamiento) en una muestra dentro del dataset; de esta manera, nos aseguramos que todas las muestras que ingresarán al clasificador serán homogéneas. Se permitirá solapamiento entre los frames de un segmento, por lo que se pueden extraer muchas más muestras de un segmento y se permitirá aumentar la tasa de clasificación cuando opere el radar, puesto que ingresarán al clasificador más muestras por unidad de tiempo.

Una vez que se obtienen todos los segmentos y frames de las señales crudas disponibles, se procede a realizar la partición de los frames en tres conjuntos: *train* (entrenamiento), *validation* (validación) y *test*. La partición **train**, como su nombre lo indica, sólo se utiliza en el entrenamiento de los modelos del clasificador, utilizándose para el

ajuste de los pesos de dichos modelos buscando la minimización de una función de costo que se define con cada modelo. La partición **validation** se utiliza cuando se realiza una optimización de los hiperparámetros de los modelos; entonces, para los distintos hiperparámetros que se usen para cada modelo, se entrena con la partición *train* y se valida con la partición *validation*. La partición de validación puede prescindirse si no se realiza optimización de modelos. Por defecto, en este trabajo no se utilizará esta partición salvo que se indique explícitamente. La partición **test** está destinada a validar el desempeño de los modelos entrenados, por lo que las métricas de desempeño que verifican los requerimientos tienen que ser las que se obtienen utilizando esta partición. La distribución de las muestras entre las distintas particiones debe ser aleatoria para mitigar sesgos, respetando una proporción (parámetro a especificar) entre la cantidad de datos para entrenamiento y para test, donde la porción de datos destinados a entrenar los modelos suele ser del 70 % al 90 % de las muestras. Es deseable que las muestras de las particiones sean independientes, pero en el caso en donde no se disponen de muchas muestras, se preferirá sólo mantener la independencia de la partición *test*. En nuestro caso, para asegurar que las muestras son independientes, se debe asegurar que no provengan de la misma adquisición (señal cruda del dataset inicial).

Sobre la partición *train*, y eventualmente sobre la partición *validation*, se aplica un proceso de **data-augmentation** (aumento de datos) que permite construir nuevas muestras a partir de las disponibles, para incrementar el tamaño de dichas particiones. En general, los modelos de clasificación entrenados de manera supervisada, tienen un mejor desempeño cuanto mayor cantidad de muestras son utilizadas para entrenarlos, especialmente mejorando la *capacidad de generalización* del modelo (desempeño sobre las muestras que no se usaron durante el entrenamiento). El aumento de datos debe realizarse conociendo la naturaleza de las señales (modelos de señal) para introducir distorsiones y combinaciones en las muestras originales, obteniendo muestras que sigan siendo representativas de dichos modelos de señal. En nuestro caso, como se trabaja con señales Doppler, el contenido frecuencial está asociado a velocidades Doppler/micro-Doppler, que pueden variar en rangos amplios. Asimismo, los patrones temporales en las variaciones frecuenciales también pueden repetirse en ciclos que varían en rangos temporales amplios. Estas dos consideraciones nos permiten comprimir o estirar las señales temporalmente para obtener nuevas muestras. Otro recurso que se utiliza normalmente en el proceso de data-augmentation es la adición de ruido, donde se puede utilizar diversos modelos de ruido que estén relacionados con la naturaleza de las adquisiciones, por ejemplo: modelos de ruido de clutter, ruido intrínseco del radar (ruido de fase, ruido térmico), ruido de fuentes externas no deseadas. En general, no se aplica este proceso a la partición de test, debido a que se prefiere que las métricas se correspondan unicamente con señales originales.

Como es posible controlar la cantidad de muestras que se generarán en el proceso de data-augmentation, se aprovecha para ajustar las cantidades que se generarán en función de las muestras originales que se tienen por cada clase (mapeo). Esto quiere decir, que si se tienen pocas muestras de una clase en particular, se generarán más muestras adicionales a partir de las señales correspondientes a esa clase. Este proceso se denomina **ecualización del dataset**, y generalmente mejora el desempeño de los clasificadores de múltiples clases, disminuyendo también el sesgo hacia algunas clases en particular.

La última etapa de esta cadena, denominada **pre-processing** (pre-procesamiento), es la conversión de la señal de cada frame a una imagen (se utilizará una matriz en punto fijo para representar dicha imagen, por cuestiones de compatibilidad con los formatos de imágenes, uso de memoria y tiempo de procesamiento). Esta conversión se debe a que los modelos de clasificación utilizados en este trabajo son del tipo CNN, donde el formato de entrada es el de una imagen con uno o más canales, dicho de otra forma, un tensor en el espacio $\mathbb{R}^{m \times n \times c}$, siendo $m \times n$ la cantidad de pixeles por canal y c la cantidad de canales (colores). La imagen por cada frame se obtiene calculando su espectrograma (frecuencia Doppler vs tiempo), por lo que la firma micro-Doppler del blanco correspondiente quedará representada como patrones espaciales (frecuencial-temporal) dentro de la imagen. Cabe mencionar que existen muchas alternativas para el pre-procesamiento de la señal temporal de cada frame, por ejemplo, podrían utilizarse wavelets obteniendo un escalograma en lugar del espectrograma [32, 33, 52]; inclusive podrían no generarse imágenes y utilizar modelos que trabajan con secuencias de datos, por ejemplo las Recurrent Neural Network (Redes Neuronales Recurrentes) (RNN), [53].

A lo largo de las etapas de esta cadena se crea y actualiza una base de datos, denominada **Metadata**, que almacena la meta-información de cada muestra y los parámetros que intervinieron en cada etapa, permitiendo la trazabilidad de cada una de las muestras del dataset con los procesos que le dieron origen o la modificaron. Esta base de datos también permite la depuración y posterior análisis de muestras que despierten cierto interés.

3.3. Cadena de entrenamiento y evaluación

La cadena de entrenamiento, denominada **Training Chain**, es la encargada de entrenar los distintos modelos a evaluar, por lo que esta cadena debe ser independiente de la familia de modelos a considerar en esta aplicación. Durante el proceso de entrenamiento se ajustan los pesos del modelo seleccionado utilizando la partición *train* del dataset. Los detalles del entrenamiento de los modelos de interés se explican en el

capítulo 7, por lo que en esta sección solamente se explicará a alto nivel la cadena y sus etapas. En la figura 3.3 se muestra esta cadena.

Como se explicó en la sección anterior, el dataset está compuesto por tres particiones (o dos, sino se utiliza la partición *validation*) y es inalterable durante el proceso de entrenamiento. Cada frame es considerado como una muestra, ya sea para entrenamiento o evaluación.

La primera etapa de entrenamiento consiste en conformar un **batch** de datos (porción de las muestras, cantidad fija), y entregarlos al clasificador (modelo), para obtener el resultado de clasificación para cada una de esas muestras del batch, este batch de entrada se denomina **Samples Batch**. El resultado de la clasificación (o inferencia) es un valor de probabilidad por cada clase, por lo que el modelo debe tener tantas salidas como clases se quieran clasificar. Los pesos del modelo son inicializados con valores aleatorios que siguen una distribución particular, y es por ello que los resultados que entregará durante la primera época de clasificación serán puramente aleatorios. Los valores que toma la salida suelen estar normalizados, limitados al rango $[0, 1]$ para que represente el valor de probabilidad mencionado.

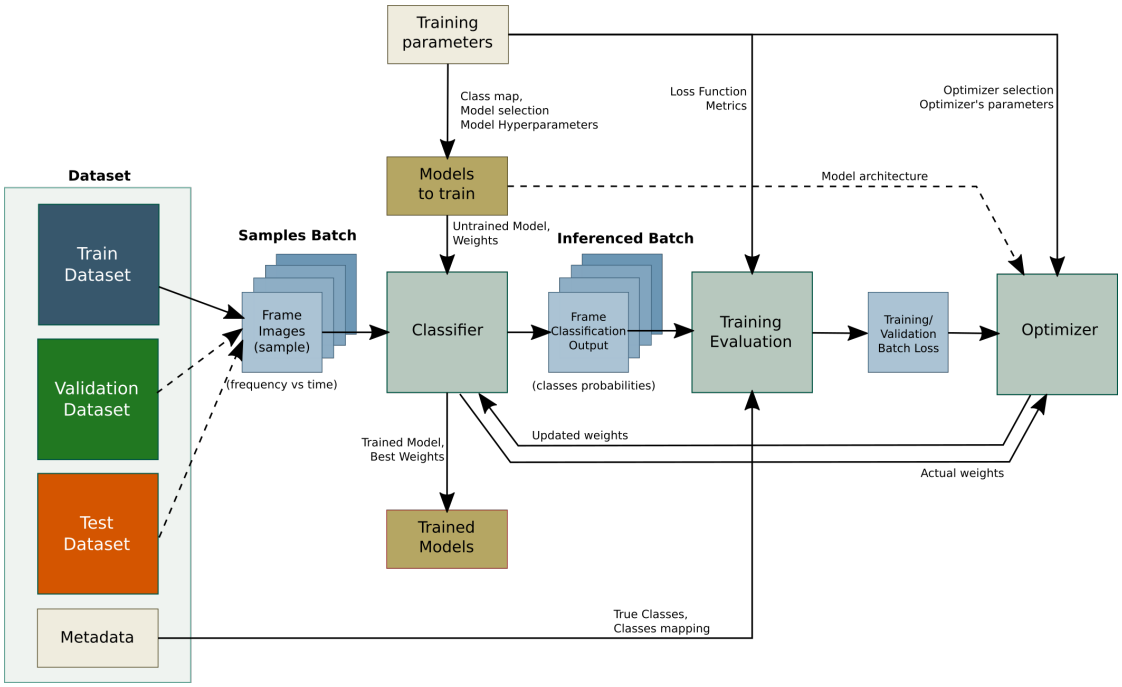


Figura 3.3: Cadena de entrenamiento - Arquitectura general.

La segunda etapa consiste en realizar la **evaluación** de los resultados de la clasificación, o **Inferenced Batch**. Esta evaluación se realiza en base a una **función de costo**, o **loss-function**, la cual da una medida del error que se cometió en la clasificación del batch. Esta función tiene, entonces, dos entradas: el vector con las predicciones y el vector con los valores verdaderos o **labels**, ambos pertenecientes al espacio $\mathbb{R}^{N_{\text{classes}}}$;

entregando como salida un valor relacionado a la disimilitud entre ambos vectores. Entre las funciones de costo más utilizadas en estas aplicaciones se pueden encontrar: *Categorical Cross-Entropy* (CCE), *Mean Squared Error* (MSE) y *Mean Absolute Error* (MAE). Aparte de la función de costo, durante la etapa de evaluación se calculan las **métricas de desempeño**, o **metrics**; siendo la variedad aún más grande, pero que se pueden destacar como más utilizadas: *Accuracy*, *Precision*, *Recall* y *MSE*. Para más información ir al capítulo 7.

Las etapas de inferencia y evaluación se ejecutan también para las otras particiones del dataset, calculándose el error y las métricas que se deseen (generalmente las mismas que se usaron para el entrenamiento). Es importante monitorear las métricas para estas particiones porque se pueden automatizar comportamientos en función de los valores que se van dando en cada ciclo del entrenamiento. Uno de los automatismos puede ser abortar el entrenamiento cuando se ha alcanzado un valor satisfactorio para las particiones de evaluación, o bien cuando las métricas para estas particiones comienzan a empeorar debido al sobreajuste del modelo (overfitting).

La tercera etapa es la de **optimización**, en donde se tiene como entrada el error entregado por la función de costo a un algoritmo denominado **Optimizer** (Optimizador), que se encarga de modificar los pesos del modelo en función del gradiente que se calcula a partir de la función de costo elegida, de modo que el ajuste de los pesos asegure que se reducirá el valor de error de dicha función para los mismos datos del batch con el que se hizo la inferencia, en resumen, se realiza una *minimización de la función de costo mediante el descenso por el gradiente*. El Optimizador calcula el gradiente del error en función de los parámetros del modelo (pesos) a través del método de **backpropagation** considerando todo el batch, luego, el error a la salida de la función de costo se va propagando a la capa previa (salida del modelo), luego a la última capa del modelo, y así sucesivamente hasta llegar a la entrada. Una vez que la sensibilidad del gradiente a cada uno de los pesos es calculada, el Optimizador define una estrategia de modificación de cada uno de los pesos en función de lo calculado. Esto asegura que el error se moverá hacia un mínimo (que puede ser local o global) vía sucesivas iteraciones. Existen diversas estrategias que el Optimizador puede emplear para acelerar la convergencia y para no estancarse en mínimos locales. Entre los optimizadores más utilizados se pueden nombrar: *Stochastic Gradient Descent (SGD)*, *Adam*, *Adadelta*, *RMSprop*.

El proceso descripto a nivel batch se repite nuevamente, conformando un nuevo batch de muestras para la nueva iteración. Este ciclo se repite hasta completar todos los batch del dataset de entrenamiento, o sea, haber utilizado todas las muestras de entrenamiento. El ciclo completo de inferencia, evaluación y ajuste de los pesos del modelo para todos las muestras de entrenamiento se denomina **época** (epoch). Generalmente,

los resultados que se muestran en las gráficas de monitoreo es a nivel época y no a nivel batch. Una sesión de entrenamiento puede tener predefinida la cantidad de épocas en las que se entrenará el modelo, como también variar en función de automatismos, como se mencionó anteriormente.

Una vez que el entrenamiento finaliza, se pueden realizar análisis de desempeño complementarios, como *Confussion Matrices (Matrices de Confusión)*, *Curvas Receiver Operating Characteristic (Característica de Operación del Receptor) (ROC)*, etc. Si los resultados son satisfactorios, se exporta el modelo entrenado. El proceso de exportación consiste en almacenar el conjunto de pesos para los que el modelo presentó el mejor desempeño en la métrica de interés, generalmente sobre la partición de *test*.

3.4. Framework

El entorno de desarrollo se construyó enteramente en **Python** (version 3)[54]. Se eligió Python ya que la mayoría de los proyectos de ML de código abierto se desarrollan en este lenguaje. Adicionalmente, se utilizaron dos librerías para redes neuronales muy poderosas : **Keras**[55] y **Tensorflow**[56], disponibles en este lenguaje.

Las cadenas funcionales han sido codificadas como scripts de Python y se ejecutan desde una terminal de comandos. Todas las cadenas se encuentran parametrizadas usando archivos JavaScript Object Notation (Notación de Objeto de JavaScript) (JSON), lo que facilita la configuración de las cadenas sin necesidad de modificar el código en los scripts, y permitiendo almacenar las configuraciones para reproducirlas posteriormente. Esta metodología ha facilitado enormemente la experimentación y ajuste de los parámetros necesarios de cada cadena.

3.4.1. Definiciones

Es importante introducir algunas definiciones relacionadas al framework que se utilizarán a lo largo del documento.

script : programa que ejecuta una secuencia de comandos o algoritmo, que en el marco de este desarrollo, se corresponden con programas a ejecutarse en Python. Las cadenas funcionales se implementan como scripts.

module : o **módulo**, es un archivo de Python que contiene funciones y/o clases que se utilizan en los scripts.

configuration : o **configuración**, es un conjunto de datos (etiquetas y valores) que parametrizan un script. Por defecto se construyen en formato JSON.

model : o **modelo**, es el conjunto de datos que definen un modelo (antes y luego del entrenamiento). En general este conjunto de datos está conformado por un script de Python que genera el modelo, especificando la arquitectura, la inicialización de los pesos, la función de costo con que se evaluará, el optimizador y las métricas de interés. Estos datos (salvo la arquitectura) se pueden modificar en tiempo de ejecución si se quiere.

session : o **sesión**, define un paquete de configuraciones y modelo particular, con un entorno independiente dentro del framework. Esto permite realizar experimentos independientes y almacenar las configuraciones y resultados, sin alterar otros experimentos realizados anteriormente. También se pueden clonar y exportar. En general, se utilizan dos tipos de sesiones: *construcción de un dataset*, o *entrenamiento/evaluación de un modelo*; pero pueden construirse otros tipos en función de las cadenas y configuraciones que se utilicen.

3.4.2. Carpetas y archivos

A nivel proyecto, todo el framework se encuentra en la carpeta **MicroDopplerRadarClassification** (repositorio). En esta carpeta se encuentran los scripts y módulos principales y una estructura de carpetas que ordena determinados tipos de archivos. A continuación se lista brevemente la estructura de carpetas:

- **root**: scripts y módulos principales para la ejecución de las cadenas funcionales.
- **configurations**: almacena configuraciones que se ejecutarán para la sesión por defecto. El framework puede resolver el path a esta carpeta usando *$CONFIG_PATH*.
- **filter**: contiene archivos que definen los filtros que se utilizan en algunas cadenas o etapas, por ejemplo: la de pre-procesamiento.
- **models**: contiene modelos. Están organizados por el mapeo que utilizan (clases a la salida del clasificador), por lo que se pueden encontrar modelos con la misma arquitectura pero para clasificar distintos mapeos. Los modelos entrenados se colocan dentro de la carpeta *trained*, también organizada por mapeo. El framework puede resolver el path a esta carpeta usando *$MODELS_PATH*.

- **outputs**: destinada a albergar los resultados de la ejecución de los scripts. El framework puede resolver el path a esta carpeta usando *$OUTPUTS_PATH*.
- **results**: destinada a exportar resultados de la ejecución de los scripts que no quieren perderse, ya que los resultados que se guardan en outputs, pueden sobreescribirse durante una nueva ejecución de los scripts.
- **sessions**: cada carpeta que se encuentra en esta ubicación define una sesión. Cada sesión tiene una réplica de la estructura de carpetas de la sesión por defecto. El framework puede resolver el path a la sección activa usando *$SESSION_PATH*.
- **support**: contiene archivos para la recuperación de entornos de trabajo, bibliotecas, módulos y lo que sea necesario para (re) establecer un entorno de trabajo operativo.
- **test**: contiene scripts que se utilizan para depurar y verificar el funcionamiento de módulos o scripts.
- **tools**: contiene scripts y (Jupyter) notebooks que sirven de herramientas para ejecutar distintas tareas del usuario dentro del framework, por ejemplo: crear una sesión, clonar una sesión, visualizar resultados de una sesión, comparar resultados entre sesiones.

El framework puede resolver el path a la carpeta principal del repositorio usando *$WORK_PATH*.

Los datos del dataset (crudos o procesados) se prefiere mantenerlos fuera de la estructura de carpetas del repositorio, para evitar que éste tenga un tamaño gigantesco. Se define, por defecto, que el dataset crudo y los dataset generados se ubican en la carpeta **data**, al mismo nivel que el repositorio en el árbol de carpetas. Sin embargo, el framework soporta trabajar en una ubicación diferente. El framework puede resolver el path a esta carpeta usando *$DATASET_PATH*.

3.4.3. Configuración y Ejecución

El procedimiento nominal para la ejecución de las tareas involucradas en las distintas cadenas se realizará ejecutando el script **main_chain.py**, cuyo archivo de configuración correspondiente es **main_chain.json**. Primero se debe parametrizar este script editando su archivo de configuración, a continuación un ejemplo:

```
{
"name": "main_chain",
```

```
"map_sel": 4,
"new_session": true,
"dataset_enable": true,
"dataset_config_file":
        "$SESSION_PATH/$CONFIG_PATH/dataset_chain.json",
"data_augmentation_enable": true,
"data_augmentation_config_file":
        "$SESSION_PATH/$CONFIG_PATH/data_augmentation_chain.json",
"preprocessing_enable": true,
"preprocessing_config_file":
        "$SESSION_PATH/$CONFIG_PATH/preprocessing_chain.json",
"train_enable": true,
"analysis_enable": true
}
```

El campo *map_sel* indica el mapeo de clases deseado (se detallará más adelante). El campo *new_session* (bool) indica si hay que descartar resultados anteriores (valor *true*) o si se debe continuar con una ejecución anterior (valor *false*). Luego se observan los campos que habilitan los scripts a ejecutar, junto a los archivos de configuración que se aplicarán a dichos scripts.

Para ejecutar las cadenas habilitadas simplemente se ejecuta en una terminal:

```
$ python main_chain.py
```

Esta ejecución también puede parametrizarse desde línea de comandos, por ejemplo:

```
# Ejecuta la sesión sandbox y fuerza una nueva session
$ python main_chain.py -s sandbox -ns

# Ejecuta la sesión principal usando un archivo de config. particular
$ python main_chain.py -cf main_default.json
```

La ejecución de las cadenas se puede realizar individualmente a través de la ejecución del script correspondiente. Por ejemplo, para ejecutar el script de entrenamiento se debe editar el archivo de configuración *train_chain.json* y luego ejecutar lo siguiente en una terminal:

```
$ python train_chain.py
```

A continuación se muestra un ejemplo del archivo *train_chain.json*:

```
{
"name": "train_chain",
"dataset_file": "../data/SPL_work/map4/SPL_corona",
"dataset_file_type": "hdf5+npz",
"log_file": "$SESSION_PATH/$OUTPUTS_PATH/train_chain_log.npy",
"tensorboard_path": "$ABS_PATH/$SESSION_PATH/$OUTPUTS_PATH/",
"map_sel": 4,
"standarization_method": "minmax",
"standarization_bias": 0.0,
"standarization_scale": 1.0,
"model_name": "model_pollito",
"new_session": true,
"batch_size": 64,
"epochs": 200,
"overwrite_compilation": true,
"loss": "categorical_crossentropy",
"optimizer": "Adadelta",
"optimizer_params": {
        "learning_rate": 0.1
        },
"metrics": ["accuracy"],
"early_stopping_enable": true,
"early_stopping_monitor": "val_accuracy",
"early_stopping_patience": 30,
"checkpoint_monitor": "val_accuracy",
"tensorboard_enable": true,
"tensorboard_histogram_freq": 10,
"remote_monitor_enable": false,
"remote_monitor_root": "http://localhost:9000",
"seed": 0
}
```

Como se mencionó anteriormente, para cubrir los experimentos de este trabajo, se ejecutan dos tipos de sesiones independientes, una es la creación de los datasets y la otra es el entrenamiento de los modelos. El nombre del dataset resultante se especifica dentro de la configuración apuntada por el campo *preprocessing_config_file*. Para **crear un dataset**, la configuración principal deshabilita las cadenas de entrenamiento y evaluación, por ejemplo:

```
{
"name": "main_chain",
"map_sel": 4,
"new_session": true,
"dataset_enable": true,
"dataset_config_file": "$SESSION_PATH/$CONFIG_PATH/dataset_chain.json",
"data_augmentation_enable": true,
```

```
"data_augmentation_config_file":
        "$SESSION_PATH/$CONFIG_PATH/data_augmentation_chain.json",
"preprocessing_enable": true,
"preprocessing_config_file":
        "$SESSION_PATH/$CONFIG_PATH/preprocessing_chain.json",
"train_enable": false,
"analysis_enable": false
}
```

Para **entrenar un modelo**, en general se parte de un dataset ya construido, por lo que se deshabilitan las cadenas involucradas en la generación de dicho dataset. El dataset se especifica dentro de la configuración apuntada por el campo *train_config_file*. Un ejemplo de la configuración sería:

```
{
"name": "main_chain",
"map_sel": 4,
"new_session": false,
"dataset_enable": false,
"data_augmentation_enable": false,
"preprocessing_enable": false,
"train_enable": true,
"train_config_file": "$SESSION_PATH/$CONFIG_PATH/train_chain.json",
"analysis_enable": true,
"analysis_config_file": "$SESSION_PATH/$CONFIG_PATH/analysis_chain.json"
}
```

3.4.4. Resultados y visualización

Durante la ejecución de los scripts, se van imprimiendo mensajes por la terminal mostrando muchos de los resultados condensados. Los resultados, dependiendo de cada script, se almacenan como archivos de texto separados por coma (CSV), archivos binarios con arreglos de datos compatibles con el paquete *Numpy* de Python, o bien como imágenes (formato png). Los resultados se alojan, por defecto, en la carpeta *outputs* de la sesión correspondiente. Si los mismos desean conservarse, es conveniente que se copien a otra carpeta (preferentemente la carpeta *results*), para no ser sobrescritos por una nueva ejecución de la sesión.

Se desarrolló un script para correr una aplicación de visualización como servidor web, llamada **sessions viewer**, que permite visualizar y comparar resultados de las distintas sesiones simultáneamente. Esta aplicación también tiene la ventaja de poder accederla remotamente, lo que facilita la visualización de datos cuando se usa otra PC o cluster para entrenar los modelos. Para ejecutar esta aplicación se debe ejecutar en la terminal:

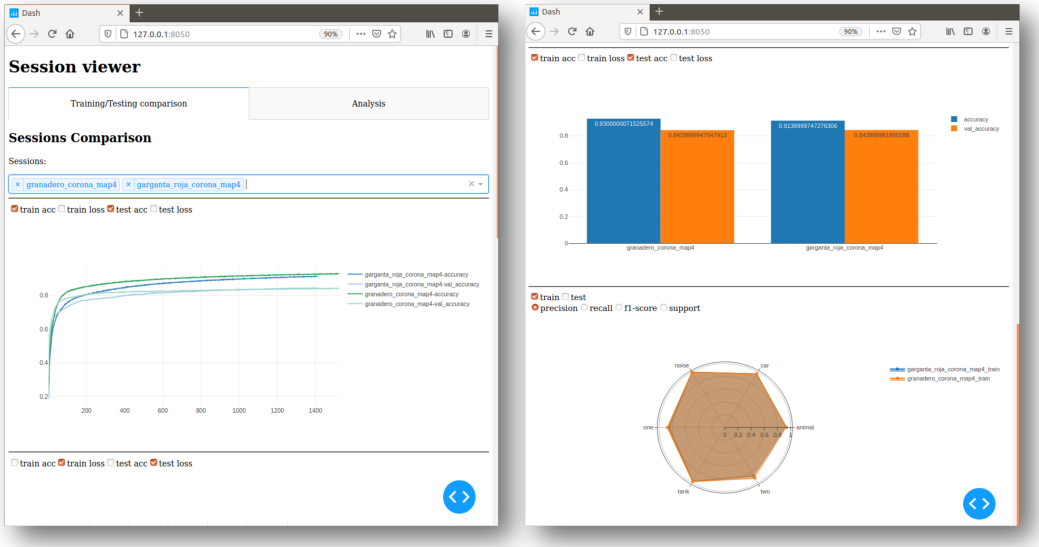


Figura 3.4: Capturas de la aplicación sessions viewer.

```
$ cd tools/
$ python ./sessions_viewer.py
```

Luego, desde cualquier navegador se debe acceder a *http://127.0.0.1:8050/*, o la dirección que se indique en la terminal. En la figura 3.4 se muestran algunas capturas de pantalla de esta aplicación.

Otra de las herramientas usadas durante los experimentos es **tensorboard** [57]. Esta es una aplicación, que también corre como servidor, que es utilizada con *tensorflow*[56] y *keras*[55] (librerías usadas para la definición y entrenamiento de los modelos). Cada vez que se ejecuta una sesión en donde está habilitado el entrenamiento de un modelo, y siempre que se haya habilitado el uso de esta herramienta (campo *tensorboard_enable* en el archivo de configuración de *train_chain*), se almacena información sobre dicha sesión, arquitectura del modelo, evolución de los pesos, y valores de las métricas en función de las épocas. Por la terminal, el script *train_chain.py* indica qué comando se debe ejecutar en otra terminal para iniciar la aplicación *tensorboard*; a continuación se muestra un ejemplo:

```
$ tensorboard --logdir=./sessions/pollito_corona_map4_adamax/outputs/
```

La dirección por defecto en donde se puede acceder a tensorboard mediante un navegador es *http://localhost:6006/*. En la figura 3.5 se muestran algunas capturas de pantalla de esta aplicación.

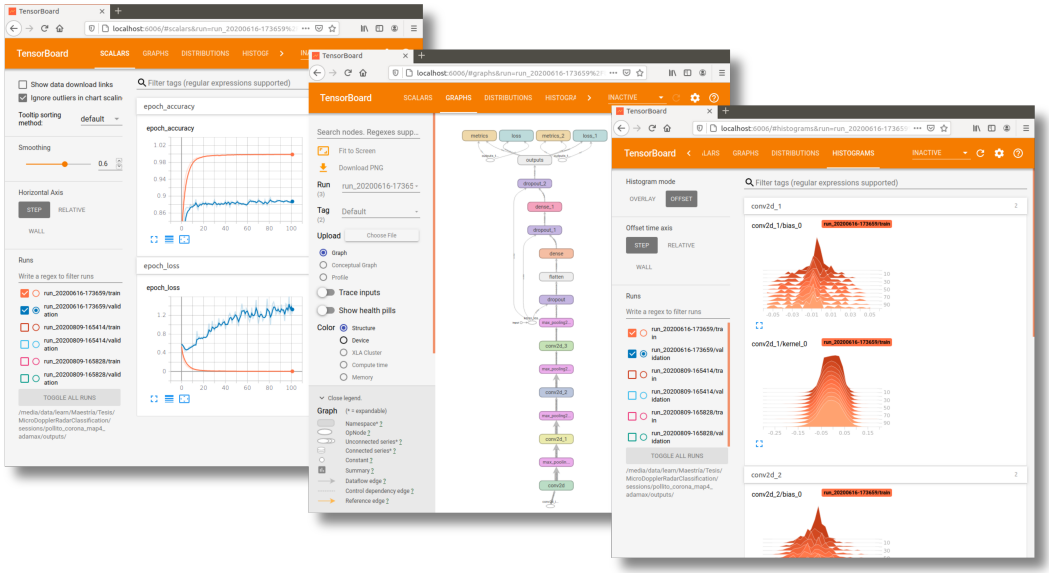


Figura 3.5: Capturas de la aplicación tensorboard.

Capítulo 4

Adquisición de datos

En este capítulo se realiza un análisis de los datos crudos obtenidos para el desarrollo del trabajo. El análisis explora cómo se encuentran categorizadas las adquisiciones realizadas para diferentes blancos, en distintos escenarios y propiedades de los mismos. El análisis persigue fundamentalmente entender las características de las adquisiciones que definirán el procesamiento de datos necesario para poder conformar uno o más datasets útiles para el entrenamiento y evaluación de los modelos de clasificador.

Estos datos se corresponden con adquisiciones de un radar particular, por lo que no son datos simulados. No se desestima expandir, en trabajos futuros, estos datos con adquisiciones de otros radares o con datos artificialmente generados a partir de un modelado de escenarios, radares y tipos de blancos.

4.1. Señales adquiridas

El dataset público encontrado, que se tomará como punto de partida (dataset raw), se denomina **The data base of radar echoes from various targets** provisto por el *Signal Processing Laboratory* del *Electrical and Computer Engineering Department* de la *Ben Gurion University* [39]; pero que para los fines de simplificar la referencia, se denominará **SPL dataset**.

Las adquisiciones del dataset se realizaron haciendo tracking sobre el blanco de interés y luego se procesaron los datos para entregar una señal correspondiente al Doppler del blanco por cada adquisición. Todas las adquisiciones se realizaron en ambientes terrestres controlados (los blancos han sido ubicados en lugares predeterminados y se ha definido arbitrariamente su movimiento), en presencia de clutter terrestre.

4.1.1. Sensor radar

Por lo extraído de [40], el radar utilizado para la adquisición de los datos es un radar de tierra para vigilancia del tipo Doppler-Pulsado. Con las siguientes características:

- Frecuencia de operación: 9 GHz.
- Ancho de banda de recepción: 3 MHz.
- Potencia pico de transmisión: 5 W.
- Arreglo de antenas tipo planar con una ganancia de 31 dB.
- Ancho de pulso: 12 μs.
- Ciclo útil: 10 %.
- Resolución en rango: 125 m.
- Resolución en acimut: 4°.
- PRF: 5681.8 Hz.

Los blancos se encontraban en línea de vista, sin presencia de interferencias físicas, y con *clutter de superficie* presente.

4.2. Características de los blancos

El **SPL dataset** contiene ecos de los siguientes tipos de blancos:

1. **Personas**: cantidad igual a: 1, 2. Con distintas características de movimiento, ángulos y equipamiento.

2. **Vehículos**: con ruedas, vehículo con orugas. Con distintas características de velocidad.

3. **Animales**: vaca, caballo, otros.

Nota: Dentro del sitio, los links para la descarga de los archivos *Readme.doc* y *clutter.zip* no funcionan. El SPL dataset original contenía muestras de clutter que ya no se encuentran disponibles en el sito web de donde se descargó [39].

Por lo extraído de [40] se puede especificar aún más las features de los blancos correspondientes a las adquisiciones radar del dataset. De esta manera:

1. Blancos relevantes[1]

   a) Persona y grupo de personas. Se presentan combinaciones de los siguientes casos:

   - Número de personas: 1, 2 y 3.
   - Velocidad: caminando lento (2-3 km/h), caminando normal (3-5 km/h), caminando rápido (5-8 km/h) y corriendo (8-9 km/h).
   - Movimiento continuo o por partes.
   - Desplazamiento sin mover las manos.
   - Desplazamiento portando una antena larga y corta.
   - Desplazamiento en línea recta o en zigzag.
   - Ángulos de la dirección de desplazamiento respecto al radar: 0°, 15°, 30°, 45°, 60°.
   - Movimiento sincronizado y desincronizado de un grupo de personas.

   b) Vehículo

   - Vehículo con ruedas, vehículo con orugas (tanques).
   - Velocidad: lenta (10-20 km/h), normal (20-30 km/h), y rápida (30-90 km/h).
   - Ángulos de la dirección de desplazamiento respecto al radar: 0°, 15°, 30°, 45°, 60°.

[1] Se considera *relevantes* a aquellos blancos que son de interés para las aplicaciones convencionales de radar, principalmente la de vigilancia.

2. Blancos irrelevantes

   a) Animal (perro, vaca, caballo, oveja, cerdo).

   b) Clutter de vegetación (árboles, arbustos, trigal).

   c) Lluvia.

   d) Cuerpos rotantes en una ubicaciones fijas (motores, rociadores de agua).

Nota: Por la inspección realizada en la estructura de carpetas y en los archivos, no se puede realizar una asociación, en algunos casos, del tipo de blanco con las características mencionadas anteriormente. Tampoco puede asociarse algunas de las características antes mencionadas con los archivos disponibles, por ejemplo: 3 personas o distintos tipos de clutter. Entonces, el desarrollo se hará con un dataset incompleto, pero aún útil debido a la diversidad de clases y características que contiene.

4.3. Archivos de datos

Los archivos a utilizar son: *animal.zip*, *one.zip*, *two.zip*, *vehicle.zip*. Una vez descomprimidos tienen el tamaño (en disco) de 1.09 GB. Los archivos descomprimidos generan una estructura de carpetas, la cual contiene los archivos de las muestras (adquisiciones) y algunos scripts de MATLAB © para poder realizar la limpieza de las adquisiciones y extraer datos útiles para el procesamiento/clasificación. Estos scripts no fueron utilizados durante el desarrollo de este trabajo.

4.3.1. Estructura de carpetas

La estructura de carpetas que se crea al descomprimir cada uno de los archivos agrupa los archivos en función de las características mencionadas en la sección anterior con algún criterio de prioridad. Por ejemplo, se prioriza agrupar primero los ángulos respecto al radar, luego las velocidades y luego cualidades particulares en algún orden arbitrario. Es importante mencionar que no todos las grupos de muestras poseen muestras que cubran todas las características, por lo que el dataset es incompleto y desbalanceado en cuanto a clases y cantidades de muestras por clase.

Los archivos que contienen los datos de las adquisiciones tienen extensión **MAT** (formato definido por el software MATLAB ©).

4.3.2. Formato de los archivos

Cada uno de los archivos MAT válidos contiene (al menos) dos variables: x y PRF. La variable x contiene los datos de la señal Doppler del blanco en formato *doble-precisión*, los datos entregados son reales (alguna de las componentes de las señales en banda base), por lo que no se podrá reconstituir la fase (como sí es posible cuando se tienen datos complejos). Estos datos fueron muestreados con la PRF del radar informada en tal variable.

4.4. Clases

Una clase, o *class*, (de un blanco) definirá el conjunto de características (features), o propiedades particulares, que lo diferencia de otros tipos de blancos en el dataset. Dentro del dataset, a cada muestra se le asociará una etiqueta que resume el conjunto de features. De acuerdo a cómo se agrupen las features disponibles en el dataset, se definirá la cantidad de clases a discriminar con el algoritmo de clasificación; a este agrupamiento se lo denomina **mapeo**.

El proceso de mapeo consiste en definir qué conjunto (combinación) de features de un archivo definirá la etiqueta (label), que luego será utilizada por el clasificador para identificar la clase a la que pertenece la muestra. Para un mismo dataset se podrá conformar más de un mapeo en función de la clasificación que se quiera realizar. Los mapeos que se utilizarán serán detallados en la sección 5.2.

4.4.1. Features

De acuerdo a la estructura de carpetas obtenida inicialmente, luego de la descompresión de los archivos zip, se puede conformar el siguiente listado de features asignables a cada archivo:

1. **classes**: animal, person, vehicle.
2. **subclasses**: car, cow, horse, noise, one, safari, tank, two.
3. **angles**: 0, 15, 30, 45, 60.
4. **speeds**: fast, normal, run, slow.
5. **directions**: go_away, jumps, line, zigzag.
6. **hands**: no_hands, strong_hands.

7. **feet**: strong_feets.

8. **equipments**: long, short.

9. **misc**: cars_on_the_road, dilugim, mitganev, syncrony, zanav.

Las etiquetas (labels) por campo son exactamente los nombres de las carpetas en donde se encuentran los archivos. Como no se tiene *metadata* de cada archivo, las características de cada blanco se extraerán de estas etiquetas. No todas las adquisiciones pueden asociarse con todas las características mencionadas anteriormente; por ejemplo, una adquisición puede no especificar la característica *hands*, entendiéndose que hay un valor "normal", o por defecto, para esta característica que no es *no_hands* o *strong_hands*.

Observación: El único cambio manual de la estructura de carpetas original fue crear la carpeta *person* para mover allí las carpetas *one* y *two*.

4.5. Análisis de las señales adquiridas

4.5.1. Cantidad y duración de las muestras

El primer análisis que se realiza es el de la cantidad de muestras que hay por cada clase. Si bien a esta altura aún no se ha especificado qué mapeo (conjunto de features por clase) se utilizará, se agrupan las muestras como las clases: *one (person), two (persons), car, tank, animal (cow, horse, safari)*. La cantidad de adquisiciones que se realizaron para cada caso se muestra en la figura 4.1.

Desde el punto de vista de las adquisiciones por clase, el conjunto de datos disponible es muy desbalanceado, teniendo muy pocas adquisiciones para la clase *tank* y muchas adquisiciones que corresponden a la clase *person* (especialmente para una persona). En principio, el desbalance puede justificarse en el hecho de que las adquisiciones realizadas para personas, tienen una gran variedad de combinaciones de features, tales como angles, speeds, directions, hands, equipment, descriptas en 4.4.

Tener en cuenta que se denomina *adquisición* a la señal que se obtuvo al iluminar un blanco particular, bajo las condiciones que correspondan, de manera continua. Las adquisiciones realizadas tienen duraciones distintas, por lo que es útil analizar la duración total del conjunto de adquisiciones por clase. Los resultados se muestran en la figura 4.2, siendo las distribuciones de las duraciones las que se muestran en la figura 4.3. Las distribuciones muestran que para las adquisiciones de personas se tienen duraciones más cortas en general, a diferencia de las otras clases, en donde lo usual es

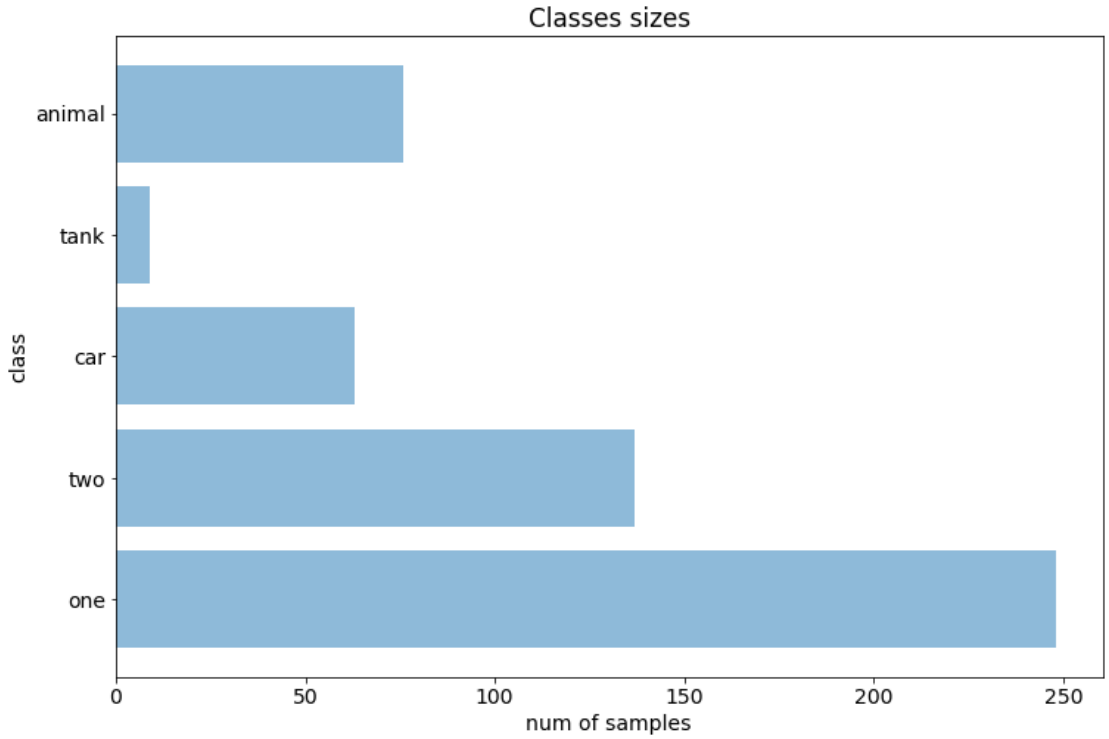


Figura 4.1: Cantidad de adquisiciones por clase (según mapeo #2).

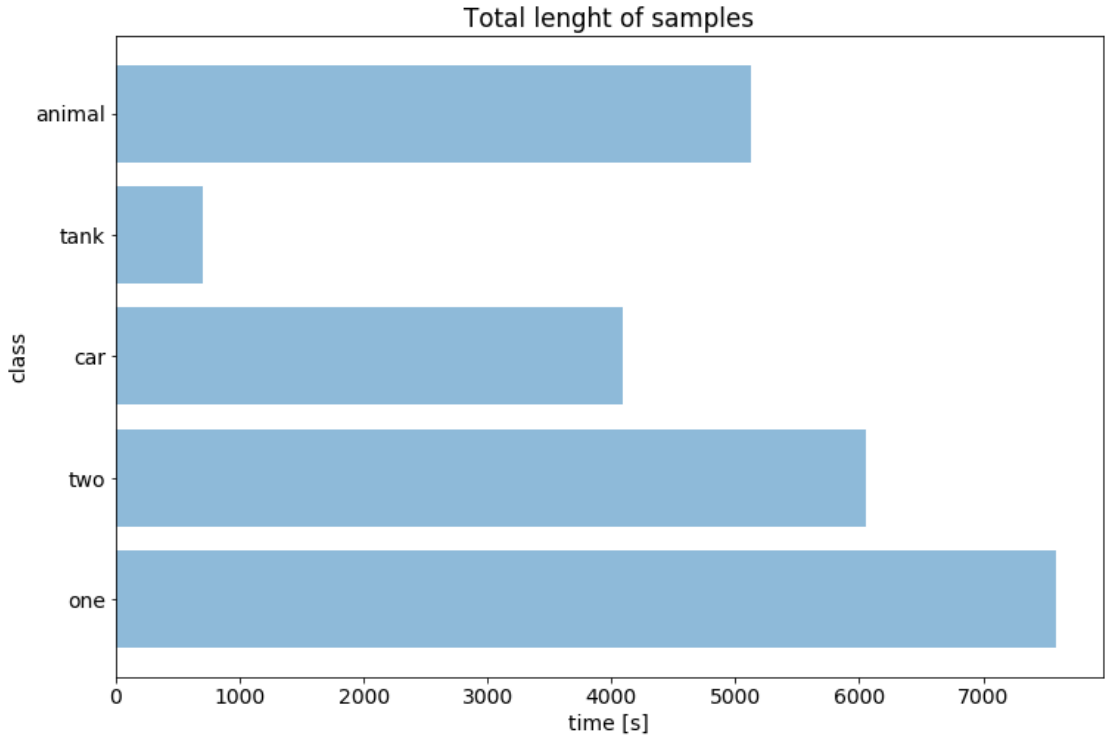


Figura 4.2: Duración total de las adquisiciones por clase (según mapeo #2).

que las adquisiciones sean prolongadas. Notar que ninguna de las adquisiciones dura menos de 5 segundos, ni más de 90 segundos.

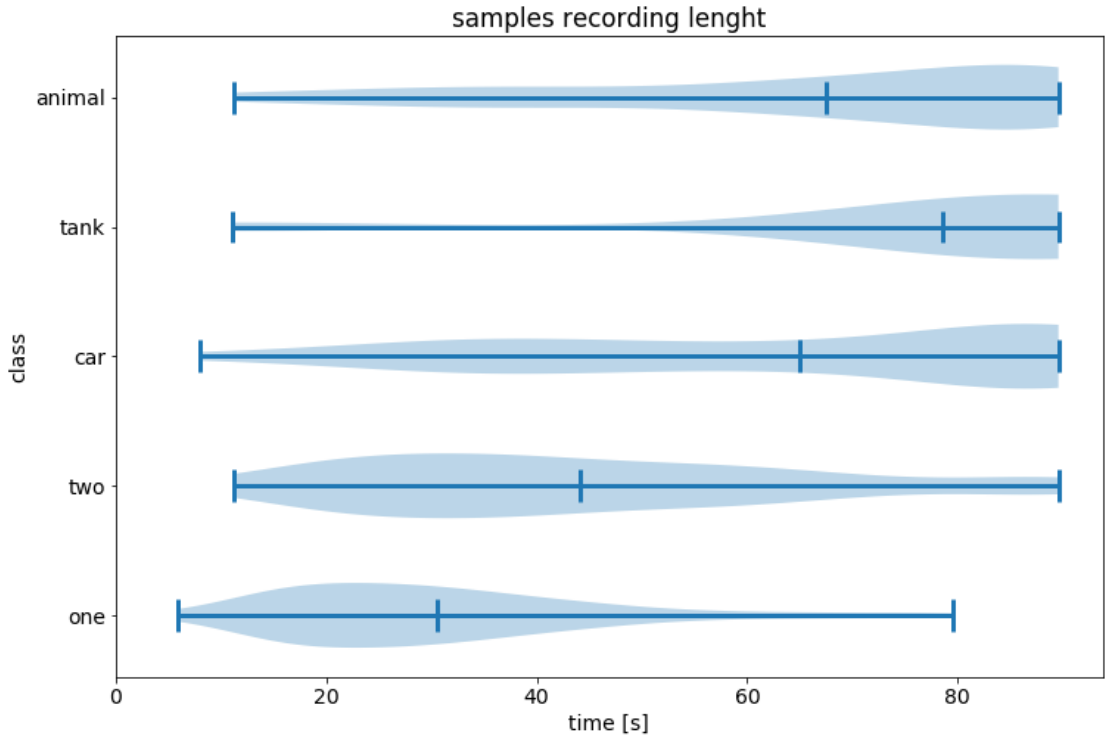


Figura 4.3: Distribuciones de las duraciones de las adquisiciones (según mapeo #2). Los segmentos verticales indican el valor mínimo, media y máximo.

4.5.2. Inspección de espectrogramas

El proceso de inspección continúa analizando el espectrograma de las señales. La intención de este análisis es simplemente visualizar los fenómenos temporales-frecuenciales que se dan para cada una de las clases, para así entender mejor el tratamiento que se realizará a los datos en las distintas etapas hasta llegar a la clasificación final. En las figuras 4.4, 4.5, 4.6 y 4.7 pueden observarse espectrogramas que se corresponden con las clases *person (one, two)*, *vehicle (car, tank)* y *animal (cow, horse, safari)*. La escala de intensidad de los espectrogramas es relativa al máximo valor de densidad espectral de potencia en toda la adquisición, salvo que se indique explícitamente otro valor de referencia.

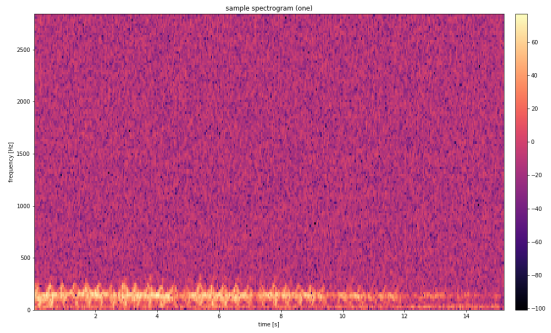
Las adquisiciones mostradas fueron seleccionadas en función de su calidad, buscando mostrar claramente el fenómeno micro-Doppler de cada tipo de blanco y la diversidad que existe dentro de cada clase. En el caso de **personas** (figuras 4.4 y 4.5) puede verse con claridad que la mayor intensidad Doppler se relaciona con el movimiento del torso, que a su vez representa la velocidad de desplazamiento media de la persona; mientras que los brazos y piernas generan las desviaciones de frecuencia periódicas. El conjunto de datos disponible presenta una variedad grande de adquisiciones con distintas características para los casos de una y dos personas. En la figura 4.4 pueden verse ejemplos de lo mencionado, donde se muestran los espectrogramas para una persona caminando

normalmente (4.4a), corriendo (4.4b), saltando (4.4c), caminando en zig-zag (4.4d), sin mover los brazos (4.4e) y arrastrándose por el suelo (4.4f). Los primeros dos casos se corresponden con los casos típicos estudiados en los modelos de micro-Doppler de personas, cuyos espectrogramas muestran una velocidad constante del torso, las componentes cuasi senoidales de piernas (mayor intensidad y menor variación frecuencial) y de los brazos (más débiles y de mayor variación frecuencial). Sin embargo, otras adquisiciones, como las restantes mostradas en la misma figura, presentarían dificultades adicionales para el clasificador, ya que tienen una variación de la velocidad del torso, componentes faltantes, o muy baja variación de frecuencia en dichas componentes. [8, 11, 43, 44]

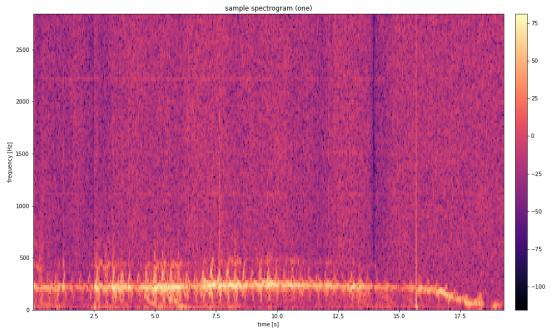
El caso en donde hay dos personas en la misma celda de resolución, el espectrograma muestra de manera superpuesta las componentes de cada persona. Notar que en la figura 4.5a se muestra el caso en el que hay dos personas caminando sincrónicamente, donde su espectrograma es muy similar al caso de una persona, lo que su clasificación representa un desafío grande. En la figura 4.5b se muestra el caso en que dos personas no caminan sincrónicamente, de aquí se puede deducir que si el espectrograma no tiene la suficiente resolución temporal, este caso puede resultar difícil de clasificar respecto al de un vehículo a baja velocidad o al de un animal.

Para el caso de **vehículos**, como los que se muestran en la figura 4.6, los espectrogramas muestran en general una única componente en frecuencia que varía con el tiempo en función de la velocidad de desplazamiento. En la figura 4.6a puede verse un **automóvil** (wheeled vehicles) a mediana velocidad con una aceleración inicial, un período de velocidad constante y luego una desaceleración. En la figura 4.6b puede verse el caso en que un automóvil frena y acelera reiteradas veces. Para el caso de **tanques** (tracked vehicles) [50], se observa oscilaciones periódicas en la componente de mayor intensidad, de la cuál no se tiene información sobre dicho comportamiento, pero una causa posible puede ser que el tanque se haya desplazado por terreno no uniforme. Sí puede observarse, más que nada en la figura 4.6d las componentes de frecuencia alrededor de la principal debido a las orugas del tanque, cuyas componentes no se observan en la de los automóviles. En general, por inspección, se observó en los espectrogramas de los tanques que la periodicidad de los patrones es menor a 4 segundos.

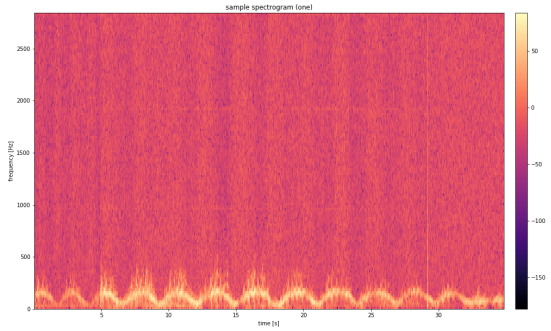
En el caso de las adquisiciones correspondiente a **animales**, como se detalló anteriormente, hay tres subclases donde la figura 4.7a muestra el espectrograma de una *vaca*, la figura 4.7b el de un *caballo* y las figuras 4.7c y 4.7d los espectrogramas de la clase *safari*. Los espectrogramas correspondiente a *vaca* y *caballo*, en general muestran frecuencias Doppler de muy bajo valor (por debajo de los $100\,[Hz]$), con algunas similitudes con los espectrogramas de personas, y menos estacionarios. En lo que respecta a la clase *safari*, los espectrogramas mayormente muestran varias componentes



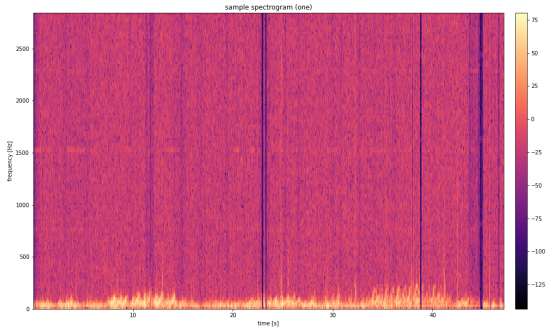
(a) Una persona.



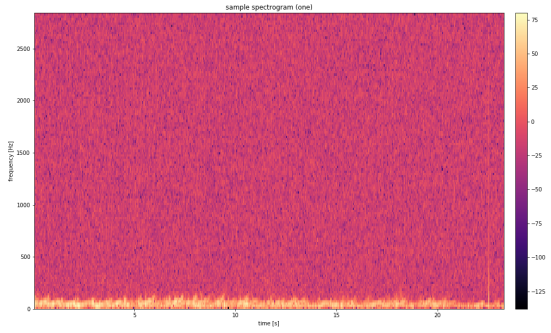
(b) Una persona corriendo.



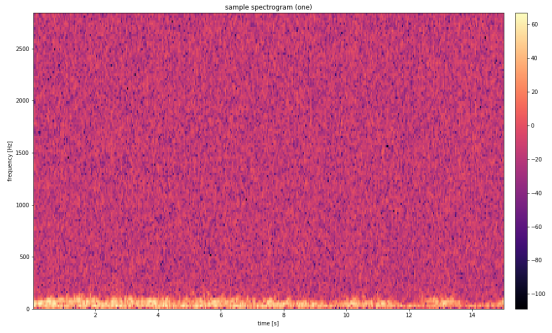
(c) Una persona saltando.



(d) Una persona caminando en zig-zag.



(e) Una persona sin mover los brazos.

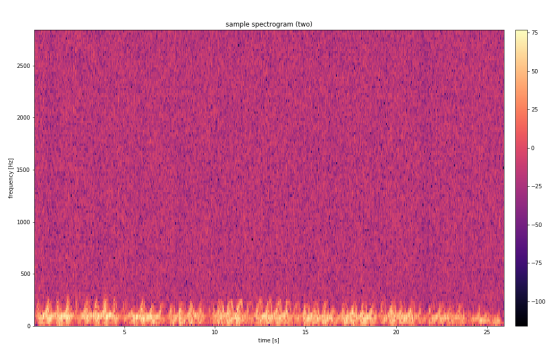


(f) Una persona arrastrándose.

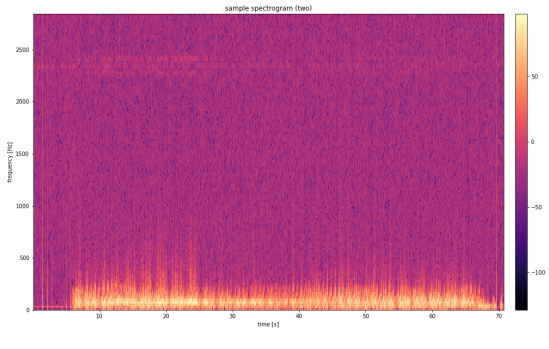
Figura 4.4: Espectrogramas de algunas adquisiciones de una persona. La escala de intensidad es relativa al máximo valor de densidad espectral de potencia en toda la adquisición.

de frecuencias con variaciones rápidas, en un espectro amplio de frecuencias, llegando en algunos casos a aproximarse a los 800 $[Hz]$; estas componentes también pueden encontrarse en simultáneo con componentes más estacionarias con un patrón similar al de la *vaca* o *caballo*. Si bien no hay detalles sobre el o los tipos de animales presentes en la celda radar, podría pensarse que se trata de aves y animales terrestres en la misma celda.

Se infiere que el conjunto de adquisiciones de animales tuvo la intención de introducir datos con el fin de analizar el error de clasificación en las otras clases, por sobre la

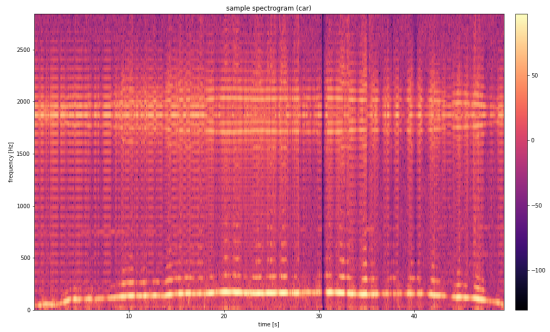


(a) Dos personas, caminando sincrónicamente.

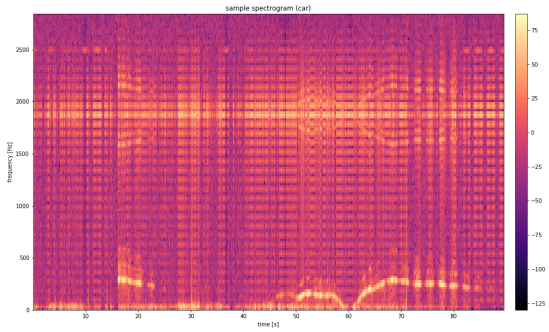


(b) Dos personas, velocidad normal, no sincronizadas.

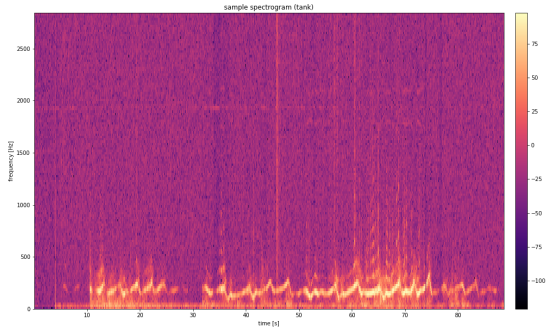
Figura 4.5: Espectrogramas de algunas adquisiciones de dos personas.



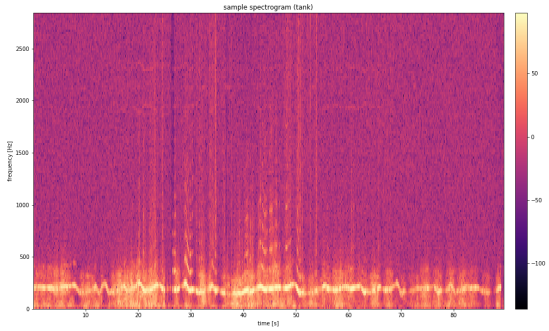
(a) Auto a baja velocidad.



(b) Auto con aceleraciones varias.



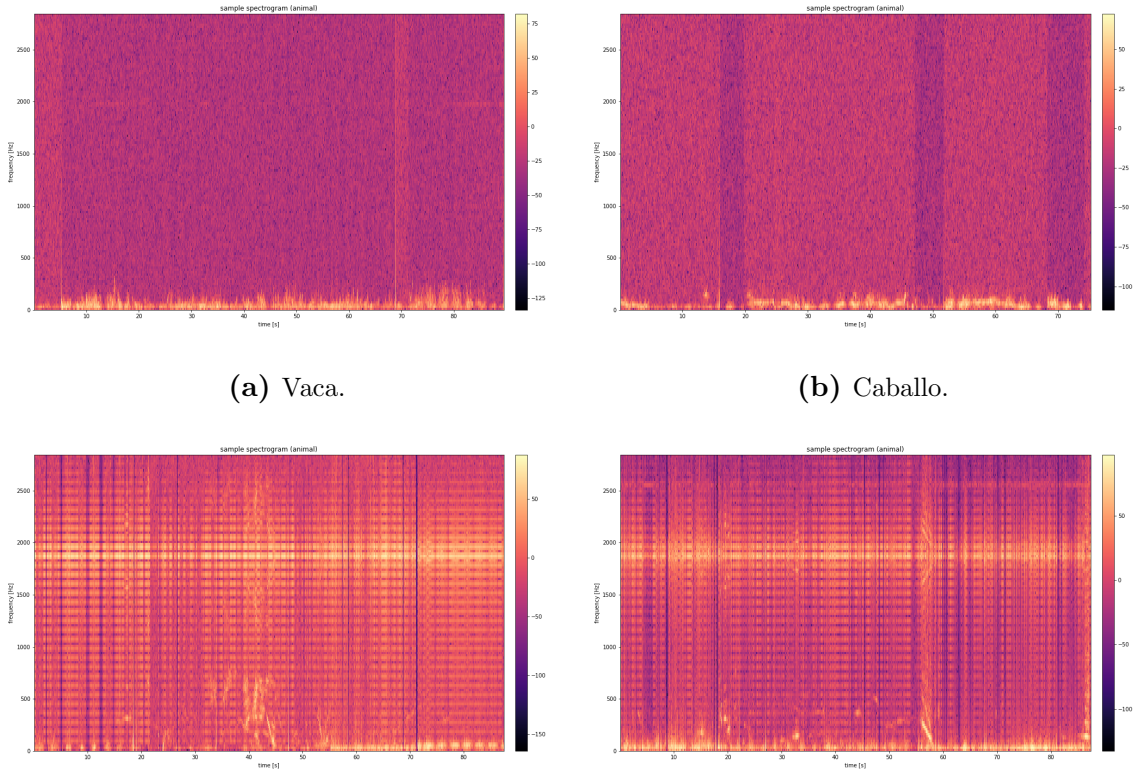
(c) Tanque



(d) Tanque

Figura 4.6: Espectrogramas de algunas adquisiciones de vehículos. Notar que en las figuras (a) y (b), el fenómeno Doppler asociado al vehículo está por debajo de los $500\,[Hz]$, el resto se corresponde a interferencias externas y/o intermodulaciones internas al radar.

clasificación en sí de estos blancos. Este tipo de blancos suelen denominarse **confusers**. Más adelante, en la evaluación de la performance del clasificador se utilizarán este tipo de clases para identificar la tasa de falsos positivos en el resto de las clases ante la presencia de animales en la celda radar.



(a) Vaca. **(b)** Caballo.

(c) Safari. **(d)** Safari.

Figura 4.7: Espectrogramas de adquisiciones de animales. Notar que en las figuras (c) y (d), el fenómeno Doppler asociado a los animales está por debajo de los 800 $[Hz]$, el resto se corresponde a interferencias externas y/o intermodulaciones internas al radar.

4.5.3. Comportamientos no deseados

Finalmente resta mostrar que existen adquisiciones, de diversas clases, en donde se encuentran comportamientos no deseados que se visualizan a través del espectrograma. Entre esos comportamientos se pueden listar a:

- **Interferencias**. La interferencia más común observada es una que ocurre a una frecuencia Doppler de aproximadamente 1,9 $[kHz]$. Esta interferencia parece deberse a algún defecto interno del radar, quizás algún acoplamiento en los osciladores locales, debido a que puede observarse en adquisiciones de diversas clases (que se asume fueron realizadas en distintos momentos y ambientes). Se observó que este comportamiento prevalece en adquisiciones de las clases *car* y *safari*, ver figuras 4.6a, 4.6b, 4.7c y 4.7d. También se puede observar en las adquisiciones de otras clases, pero con menor intensidad, como en las figuras 4.4c, 4.6c y 4.7a; o bien en otro valor de frecuencia como se observa en las figuras 4.4b, 4.4d, 4.5b y 4.6d.

- **Intermodulaciones**. Cuando aparecen las interferencias mencionadas en el pun-

to anterior, generalmente aparecen productos de intermodulación del espectro de interés con dicha frecuencia, como así también réplicas de la frecuencia de interferencia en todo el espectro Doppler definido por la PRF. Estos dos fenómenos de intermodulación pueden observarse con claridad en las figuras 4.6a, 4.6b, 4.7c y 4.7d.

- **Pérdida de señal**. En la mayoría de las adquisiciones se observa que hay lapsos de tiempo en donde se pierde la señal Doppler de interés, en algunos casos se trata de tiempos muy cortos menores a $500\,[ms]$, y en otros casos muy extensos, superando los $4\,[s]$ (tiempo en el que se requiere realizar una clasificación); en muy pocos casos se da el extremo en donde prácticamente no hay señal de interés en toda la adquisición. Este comportamiento se manifiesta de dos maneras, una es cuando se pierde la señal completamente (todo el rango de frecuencia) y no sólo lo que se correspondería con el blanco de interés; y la otra manera es cuando se pierde sólo la señal de interés (el espectro que corresponde al ruido no se altera significativamente). No se tiene información del por qué de este comportamiento, pero se supone que cuando se pierde la señal en todo el rango de frecuencias es debido a un desenganche de alguno de los osciladores locales o su referencia (en algunas adquisiciones se observa que cambia el valor de la frecuencia de interferencia cuando ocurre la pérdida de señal); mientras que cuando se pierde sólo la señal del blanco se supone a que puede haber sido un problema en el tracking del blanco.

En la figura 4.8 se muestra una adquisición en donde se encontraron diversos comportamientos no deseados, con sólo 6 segundos útiles (con presencia de la señal de interés). Si bien este es uno de los casos extremos entre las adquisiciones del dataset, estos comportamientos se presentan en muchas de las adquisiciones. Esto justifica que se realice un pre-procesamiento de la señal para poder extraer los intervalos útiles para el entrenamiento y evaluación del clasificador. Este pre-procesamiento se detallará más adelante en el punto 5.1, Depuración y extracción de segmentos/frames.

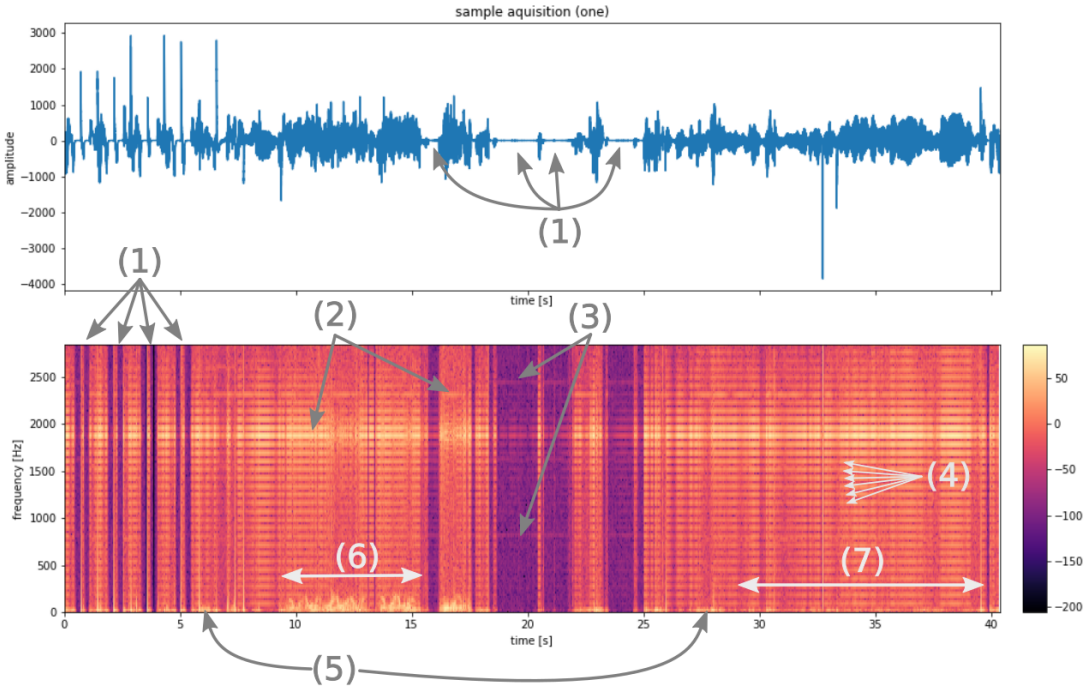


Figura 4.8: Adquisición con una diversidad de comportamientos anormales (no deseados). (1): Pérdida de señal (intervalos cortos y largos). (2): Interferencias. (3): Interferencia que se visualiza ante una pérdida de señal. (4): Intermodulaciones con interferencias. (5): Ruido tipo clutter, no se corresponde con el la señal Doppler de interés. (6): Intervalo útil para la clasificación (presencia de la señal de interés). (7): Ausencia de señal para clasificación (con interferencias presentes).

Capítulo 5

Dataset

Una vez realizada la inspección de los datos crudos, presentada en el capítulo 4, se tiene que realizar el proceso que convierta a ese conjunto de señales en imágenes que serán las entradas del clasificador. Este proceso fue presentado, a alto nivel, en la sección 3.2, denominándose **Dataset Chain**. En este capítulo se detallarán las etapas que conforman dicha cadena, mostrando algunos de los resultados intermedios y finales, que llevan a la conformación definitiva de cada dataset que se usará para el entrenamiento y evaluación de los modelos.

Es importante mencionar que, si bien se espera que los modelos que pertenecen al tipo Deep Learning aprendan las propiedades de los datos sin la necesidad de particulares algoritmos de pre-procesamiento, es necesario tener un dataset con determinadas características que hacen a la calidad y representatividad del espacio muestral, en definitiva de los datos que no conocemos y a los que estará expuesto el clasificador una vez que se haga el despliegue en el radar. De esa manera no se condiciona el desempeño de la clasificación por culpa de los datos.

Tener en cuenta que existen diversos métodos que se podrían adoptar para la conversión de las señales Doppler en datos de entrada a un clasificador, pero como se explicó en 3.2, en este trabajo se optó por elegir la conversión a imágenes de tamaño fijo que se forman a partir del espectrograma de la señal.

5.1. Depuración y extracción de segmentos/frames

Las acciones que se realizan en la primera etapa de la *Dataset Chain* están dedicadas a obtener los segmentos válidos de las señales correspondientes a todas las adquisiciones, y a partir de ellos obtener los frames. Esta etapa, como la última de la cadena, hacen uso del espectrograma de la señal. El espectrograma se obtiene a partir de la Short-Time Fourier Transform (Transformada de Fourier de Tiempo Corto) (STFT) que se aplica a la señal Doppler. Esta transformación se detallará a continuación, más con un enfoque en la parametrización que en el fundamento teórico.

5.1.1. Short-Time Fourier Transform (STFT)

Esta transformación consiste en fragmentar una señal y aplicar la transformada de Fourier a cada uno de esos fragmentos (chunks). El hecho de calcular la transformada de Fourier en fragmentos de duración corta, permite ver la evolución del contenido espectral de la señal a lo largo del tiempo. La STFT [58] es calculada aplicando una ventana deslizante de longitud W_{len} (la ventana va definiendo los fragmentos de la señal). La ventana deslizante aplica una función (window's function o función de la ventana), que denominaremos $g_W[n]$, que se destina generalmente a eliminar artefactos que aparecen por el recorte de la señal.

Se puede aplicar un solapamiento de los fragmentos (overlapping) lo que suaviza las transiciones entre cada transformada y compensa las atenuaciones introducidas por el ventaneo. Por lo general se especifica la cantidad de muestras de solapamiento entre un fragmento y el siguiente, que denominaremos N_{ov}, y define el salto en muestras $N_H = W_{len} - N_{ov}$ que es proporcional a la resolución temporal de la STFT, siendo entonces $t_{res} = N_H/f_s$ (donde f_s es la frecuencia de muestreo). Entonces, dada una señal $\boldsymbol{x}$, de $N_{\boldsymbol{x}}$ muestras, se obtendrán N_c fragmentos, siendo:

$$N_c = \left\lfloor \frac{N_{\boldsymbol{x}} - W_{len}}{N_H} \right\rfloor + 1 = \left\lfloor \frac{N_{\boldsymbol{x}} - N_{ov}}{N_H} \right\rfloor = \left\lfloor \frac{N_{\boldsymbol{x}} - N_{ov}}{W_{len} - N_{ov}} \right\rfloor \tag{5.1}$$

Notar que N_c será la cantidad de puntos que tendrá la STFT en la dimensión temporal, mientras que la cantidad de puntos que tendrá en la dimensión de las frecuencias dependerá de la transformada de Fourier que se realice. Para este tipo de aplicaciones y señales muestreadas, lo común es utilizar la Discrete Fourier Transform (Transformada de Fourier Discreta) (DFT) [59] en cada uno de los fragmentos, y por cuestiones de eficiencia lo conveniente es calcular la DFT mediante la Fast Fourier Transform (Transformada de Fourier Rápida) (FFT) [60]. Para mayor detalle ver [61].

Podemos obtener la transformada de Fourier del k-ésimo fragmento de la señal $\boldsymbol{x}$

a partir de la Discrete-Time Fourier Transform (Transformada de Fourier de Tiempo Discreto) (DTFT) aplicando una ventana g_W sobre dicho fragmento:

$$\boldsymbol{X}_k(f) = \mathcal{F}\{\boldsymbol{x}_k\} = \sum_{n=-\infty}^{\infty} \boldsymbol{x}[n] g_W[n - kN_H] e^{-j2\pi f n} \quad ; k = 0, ..., N_H - 1 \tag{5.2}$$

Desde el punto de vista práctico de la implementación podemos trabajar a nivel fragmento, y calcular la *DFT* de cada uno usando la *FFT*, por lo que:

$$\boldsymbol{X}_k[\nu] = \sum_{n=0}^{N_{FFT}-1} \boldsymbol{x}_k[n] g_W[n] e^{-j2\pi\nu n/N_{FFT}} \quad ; \nu = 0, ..., N_{FFT} - 1 \tag{5.3}$$

donde $\boldsymbol{x}_k$ es el k-ésimo fragmento de la señal $\boldsymbol{x}$, o sea $\boldsymbol{x}[kN_H : (k+1)N_H - 1]$, y donde ν representa el índice de las frecuencias (Doppler), siendo $f = \nu f_s / N_{FFT}$[1]. Los vectores de señal y ventana se completan con ceros (zero-padding) si las dimensiones de los mismos no coinciden con N_{FFT}. De la ecuación 5.3, se observa que la transformada de cada fragmento pertenece a $\mathbb{C}^{N_{FFT}}$, pero si la señal es real sólo podríamos quedarnos con la mitad del espectro más la componente continua, por lo que la transformada podría restringirse al dominio $\mathbb{C}^{N_{FFT}/2+1}$.

La *STFT* la podremos escribir entonces en función de las *DTFTs* de los fragmentos como:

$$STFT\{\boldsymbol{x}[n]\}(f) \equiv \boldsymbol{X}(f) = \begin{bmatrix} \boldsymbol{X}_0(f) & \boldsymbol{X}_1(f) & \cdots & \boldsymbol{X}_{N_c-1}(f) \end{bmatrix} \tag{5.4}$$

En esta aplicación, luego de aplicar la *DFT* a cada fragmento de la señal $\boldsymbol{x}$, se puede construir una matriz cuyas columnas serán dichas transformaciones, según 5.3, resultando la *STFT*:

$$STFT\{\boldsymbol{x}[n]\}[\nu] \equiv \boldsymbol{X}[\nu] = \begin{bmatrix} \boldsymbol{X}_0[\nu] & \boldsymbol{X}_1[\nu] & \cdots & \boldsymbol{X}_{N_c-1}[\nu] \end{bmatrix} \quad \in \mathbb{C}^{N_{FFT} \times N_c} \tag{5.5}$$

5.1.2. Espectrograma

El espectrograma [61, 62] de una señal muestra la intensidad de las componentes de frecuencia en función del tiempo. Para obtener el espectrograma basta con calcular

[1] Se supone largo de la ventana como N_{FFT}, pudiéndose usar W_{len} si no vale la suposición. Otros secuencias pueden ser evaluadas convenientemente, tales como $[-\frac{N_{FFT}}{2}, \frac{N_{FFT}}{2} - 1]$ si N_{FFT} es par, o $[-\frac{N_{FFT}-1}{2}, \frac{N_{FFT}-1}{2}]$ si N_{FFT} es impar.

la potencia de las componentes de frecuencia como módulo cuadrado, por lo que:

$$spectrogram\{\boldsymbol{x}[n]\}(t,f) = \left|STFT\{\boldsymbol{x}[n]\}(f)\right|^2 \tag{5.6}$$

Tener en cuenta que $t = n/f_s$. Esto construye una imagen, correspondiente con la matriz original entregada por la STFT (utilizando la ecuación 5.5), en donde la intensidad de cada elemento de dicha matriz (pixel imagen) se corresponde con el valor de potencia de una frecuencia en un intervalo de tiempo. Para este trabajo, se usará el eje vertical (filas) para el dominio de las frecuencias, y el eje horizontal (columnas) para el dominio temporal. Esa relación de compromiso también puede expresarse como la capacidad de visualizar eventos que presentan cambios rápidos en las componentes frecuenciales contra visualizar componentes que se encuentran muy juntas en el dominio de la frecuencia. En la figura 5.1 puede verse el espectrograma obtenido para una de las adquisiciones del dataset sin ningún procesamiento previo. Más ejemplos ya fueron mostrados en el capítulo 4.

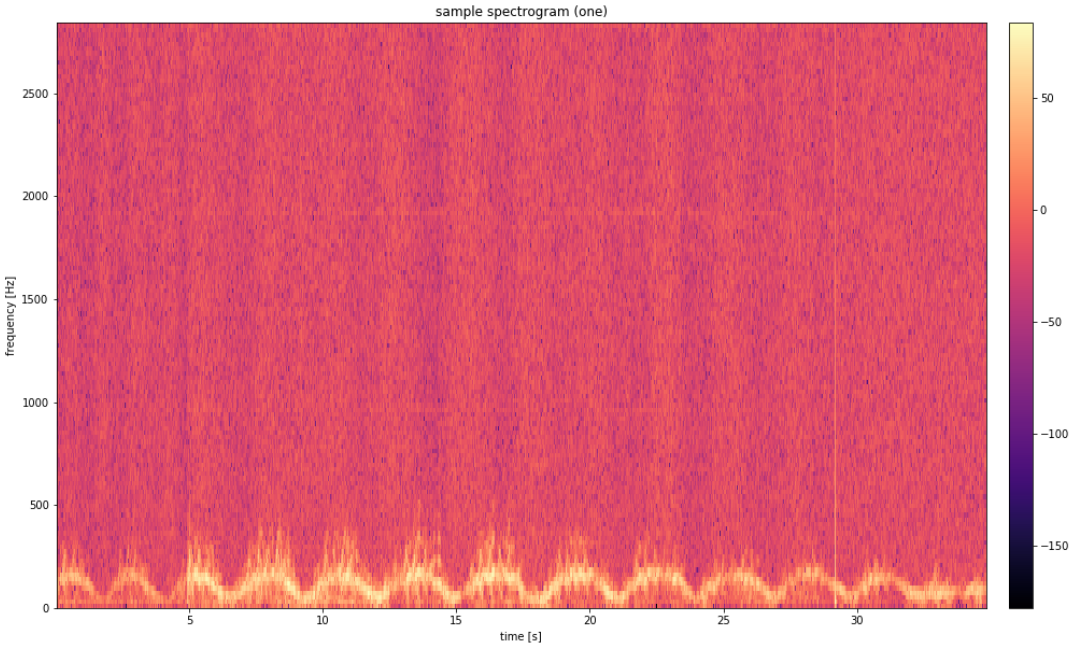


Figura 5.1: Espectrograma de la adquisición *SPL-person-one-0-dilugim-5-25* (# 90). Una persona corriendo y saltando con un ángulo de 0° respecto del radar. El eje de las frecuencias (vertical) se encuentra en Hz desde 0 a $f_s/2$ (siendo $f_s = 5681{,}81$ Hz), mientras que el eje horizontal está en tiempo (segundos). Se utilizó una ventana tipo *hamming* de 256 muestras y una FFT de igual longitud, con un solapamiento de 32 muestras. Esto nos da un salto $N_H = 224$ muestras, lo que nos da una resolución temporal $t_{res} = 39{,}4$ ms aproximadamente. La escala de intensidad es relativa al máximo valor de densidad espectral de potencia en toda la adquisición.

Por las características de la transformada de Fourier, para obtener una buena resolución en el dominio (eje) de las frecuencias, se tiene que utilizar fragmentos más grandes (W_{len} grande) lo que implica una degradación de la resolución temporal. De manera inversa, si se desea tener una resolución temporal buena hay que trabajar con fragmentos cortos, lo que degrada la resolución en frecuencia. Tenemos un parámetro adicional, que es el solapamiento entre fragmentos, mejorar la resolución temporal para fragmentos largos, a costa de compartir parte de la información entre fragmentos (muestras comunes entre fragmentos contiguos). Estos parámetros nos permitirán ajustar las resoluciones temporales y frecuenciales, para poder capturar de una manera útil los patrones de las firmas micro-Doppler de los blancos del dataset. Estos parámetros también tendrán que ser ajustados cuando se quieran obtener las imágenes de tamaño fijo a partir del espectrograma de cada frame.

5.1.3. Parametrización del espectrograma

La etapa de extracción de los segmentos útiles de las adquisiciones se parametriza definiendo la ventana a utilizar en el STFT, definiendo tipo y longitud; y la resolución temporal que se quiere tener en el espectrograma resultante. Opcionalmente se puede definir la cantidad de puntos para la FFT, pero por defecto se utiliza la cantidad de puntos igual al tamaño de la ventana. Se decide utilizar la unidad temporal y no de muestras, para independizarse de la frecuencia de muestreo que se haya utilizado en las adquisiciones. A continuación se muestra un ejemplo del fragmento del archivo de configuración de esta etapa en donde se definen estos parámetros.

```
{
"window": "hamming",
"wlen": 1024,
"t_res": 0.05,
"nfft": 1024,
}
```

A partir de estos parámetros y la frecuencia de muestreo (que se lee de los metadatos de la adquisición), se calcula la cantidad de muestras a solapar, N_{ov}, entre los fragmentos de la adquisición:

$$N_{ov}(W_{len}, t_{res}, f_s) = W_{len} - \lceil t_{res} \cdot f_s \rceil = W_{len} - N_H \tag{5.7}$$

Si el tiempo de resolución no se especificara, se toma por defecto que $N_{ov} = \lfloor W_{len}/8 \rfloor$.

5.1.4. Filtrado

En base a lo presentado en la sección 4.5.3, en algunas adquisiciones existen interferencias e intermodulaciones no deseadas distribuidas en un rango amplio de frecuencias. Considerando también que las componentes frecuenciales de la señal Doppler de los blancos del dataset se encuentran en su totalidad por debajo de los 800 Hz, se considera beneficioso filtrar todas las adquisiciones con un filtro pasa-bajos, cuya frecuencia de corte (3 dB) será de 800 Hz. El parámetro que define el filtro se especifica de la siguiente manera:

```
{
"segm_filter": "$SESSION_PATH/filters/SegmentationFilter.npz",
}
```

Notar que el filtro está especificado como un archivo de Numpy. Este archivo contiene los factores (numerador y denorminador) de un filtro FIR.

5.1.5. Método de selección de un segmento útil

En esta instancia disponemos del espectrograma de la señal correspondiente a la adquisición de interés filtrada. Para automatizar el proceso de selección de los segmentos útiles se procede a identificar los fragmentos del espectrograma cuya densidad espectral máxima supere un nivel a especificar denominado *threshold*, S_{th} (parámetro *segm_threshold* de la configuración) que se elije de manera experimental [2]. El proceso consiste en convertir el espectrograma a dB (puesto que es común trabajar con estas unidades), luego recorrer el espectrograma por fragmentos e identificar el valor de densidad espectral máximo en la dimensión de las frecuencias; de esto se obtendrá un vector con N_c valores al cual se le aplicará un suavizado utilizando una ventana deslizante g_{MW} uniforme y normalizada de largo MW_{len} muestras. Se prefiere especificar el largo de esa ventana deslizante también en unidades de tiempo, como $MW_{\text{len sec}}$ (parámetro *segm_wlen* de la configuración), por lo que la longitud de la ventana en muestras deberá calcularse como:

$$MW_{\text{len}} = \left\lfloor \frac{MW_{\text{len sec}}}{t_{res}} \right\rfloor = \left\lfloor \frac{MW_{\text{len sec}} \cdot f_s}{N_H} \right\rfloor \tag{5.8}$$

Definiendo como $\mathbf{1}_{1\times n}$ a un vector de unos de largo n, la función de la ventana uniforme normalizada se escribe como:

$$g_{MW} = \frac{\mathbf{1}_{1\times MW_{\text{len}}}}{MW_{\text{len}}} \tag{5.9}$$

[2] Se analizaron los segmentos útiles resultantes para distintos valores de *threshold* en adquisiciones particulares que presentaban diversos problemas como los mencionados en 4.5.3, eligiéndose un valor conveniente de *threshold* por inspección.

Partiendo de la ecuación 5.6, la conversión a dB del espectrograma, la matriz que se obtiene se expresa como:

$$\boldsymbol{S}_{dB}[n_t, n_f] = 10\log(spectrogram\{\boldsymbol{x}[n]\}[n_t, n_f]) = 20\log\left|STFT\{\boldsymbol{x}[n]\}[n_f]\right| \quad (5.10)$$

Aplicando el suavizado a los máximos encontramos el vector $s_{\text{psd max}}$:

$$\boldsymbol{s}_{\text{psd max}}[n] = g_{MW} * \underset{n_f}{\text{máx}}\left\{\boldsymbol{S}_{dB}\right\} \quad , n = 0, ..., N_c - 1 \quad (5.11)$$

Luego se procede a comparar los valores $\boldsymbol{s}_{\text{psd max}}$ con el *threshold*, S_{th}, obteniendo las porciones útiles de la adquisición cuando dichos valores superan el umbral. Si un fragmento tiene una $\boldsymbol{s}_{\text{psd max}}$ que iguale o supere dicho umbral, será considerado un **fragmento útil**; sin embargo no lo podemos considerar un segmento útil. Definiremos como **segmento válido** al conjunto de fragmentos útiles que cumplan una serie de condiciones:

1. **Tiempo máximo entre fragmentos inútiles**: Los fragmentos inútiles cuya duración no supere un máximo establecido por el parámetro *max_gap_time* se incluirán dentro del segmento como fragmentos útiles.
2. **Duración mínima consecutiva**: debe haber fragmentos útiles consecutivos por una cantidad mínima de tiempo (parámetro *min_segm_len* de la configuración).

El ejemplo de la figura 5.2 muestra la extracción de dos segmentos válidos de una adquisición. El valor de *threshold* se eligió por inspección visual de los espectrogramas e iteración de este proceso de selección. Notar cómo el filtrado previene de que se consideren como válidos segmentos con interferencias o ruidos fuera de banda.

Los parámetros de configuración que intervienen en este proceso son:

```
{
"segm_threshold": 25.0,
"segm_wlen": 2.0,
"max_gap_time": 0.5,
"min_segm_len": 3.0,
}
```

Este mismo proceso también puede ejecutarse para extraer segmentos que no contengan señal Doppler significativa, cambiando el criterio de selección para elegir aquellos fragmentos que se encuentren por debajo del umbral de selección. Esto puede resultar útil si se quiere construir un dataset con muestras que representen interferencias,

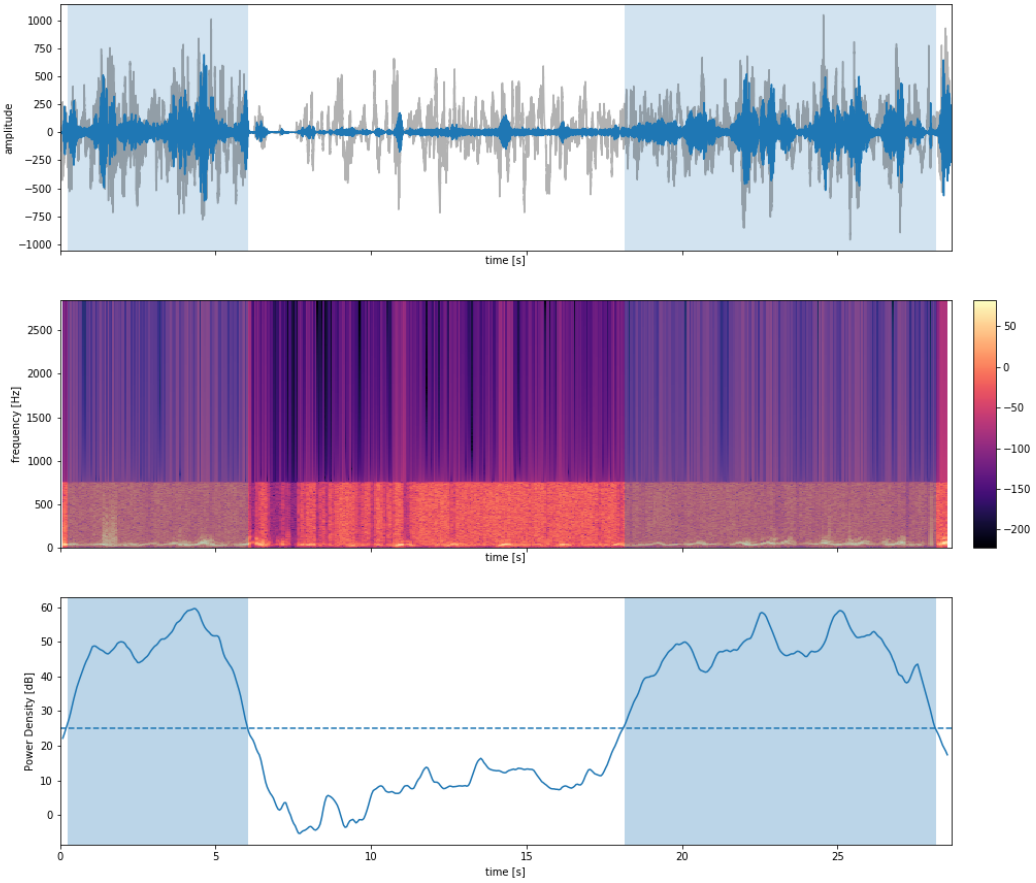


Figura 5.2: Segmentos válidos extraídos de una adquisición (resaltados en franjas azules). En el gráfico superior se observa la señal Doppler de una adquisición (gris) y la señal filtrada (azul). En el gráfico del medio se observa el espectrograma de la señal filtrada, notar el filtrado en los 800 Hz. En el gráfico inferior se observan los valores máximos de la densidad espectral de cada fragmento (línea sólida) y el valor de *threshold* (línea de trazos) que determina la selección de fragmentos útiles. En este ejemplo el valor de *threshold* es de 25 dB/Hz.

ruidos o fuentes de confusión. Este proceso es opcional y se puede habilitar mediante el parámetro *add_noise* del archivo de configuración (parámetro tipo *bool*). En el archivo de configuración se puede parametrizar este proceso mediante los siguiente parámetros:

```
{
"add_noise": false,
"segm_threshold_noise": 10.0,
"t_res_noise": 0.05,
"segm_wlen_noise": 2.0,
"max_gap_time_noise": 0.25,
"min_segm_len_noise": 3.5
}
```

5.1.6. Extracción de frames

Como se mencionó en la sección 3.1 y 3.2, un **frame** es la porción de la señal que se convertirá en una muestra destinada a entrenamiento, evaluación o inferencia. Se estableció que el frame tenga una duración fija, la cual se especifica con el parámetro *frame_len* del archivo de configuración. Por otro lado, podemos permitir un solapamiento de la señal entre frames consectutivos dentro de un segmento válido, este solapamiento lo define el parámetro *frame_overlap* del mismo archivo de configuración. A continuación se muestra el extracto del dicho archivo con los parámetros mencionados:

```
{
"frame_len": 4.0,
"frame_overlap": 3.5,
}
```

Para la **extracción de frames válidos** se toman cada uno de los segmentos válidos de las adquisiciones y se fragmenta la señal de cada segmento para cumplir con la *duración* requerida para el frame (parámetro *frame_len* del archivo de configuración) y un *solapamiento* entre frames (parámetro *frame_overlap* del archivo de configuración). Es el mismo proceso que se usa en la STFT para la extracción de los fragmentos (ver 5.1.1). Este proceso permite obtener muestras temporales de la señal Doppler que tienen información útil con una duración fija. En el caso de existir solapamiento habrá una mayor correlación entre frames cercanos. Si se quiere minimizar la correlación entre los frames de una adquisición se debe minimizar el solapamiento entre frames (que podría ser nulo en el caso extremo), lo que impactará sobre la cantidad de muestras que luego tendrá el dataset, ya que a menor solapamiento, menor cantidad de muestras.

La cantidad de frames que se extraen de un segmento depende, entonces, de la duración del segmento N_s (cantidad de muestras del segmento), la duración del frame N_f y el solapamiento entre frames N_{fov} (no confundir con el solapamiento N_{ov} utilizado para el cálculo del espectrograma). La ecuación 5.12 nos permite calcular esta cantidad de manera análoga a la ecuación 5.1 usada para calcular el número de fragmentos de un espectrograma.

$$M_f = \left\lfloor \frac{N_s - N_f}{N_{fH}} \right\rfloor + 1 = \left\lfloor \frac{N_s - N_{fov}}{N_{fH}} \right\rfloor = \left\lfloor \frac{N_s - N_{fov}}{N_f - N_{fov}} \right\rfloor \tag{5.12}$$

Siendo M_f la cantidad de frames que se extraen del segmento y N_{fH} la cantidad de muestras entre el inicio de frames consecutivos. Recordar que el número de muestras y la duración están relacionadas a través de la frecuencia de muestreo f_s, por lo que la duración de un frame en segundos sería $T_f = N_f/f_s$.

Por defecto se elige una duración de 4 segundos para cada frame debido a que con

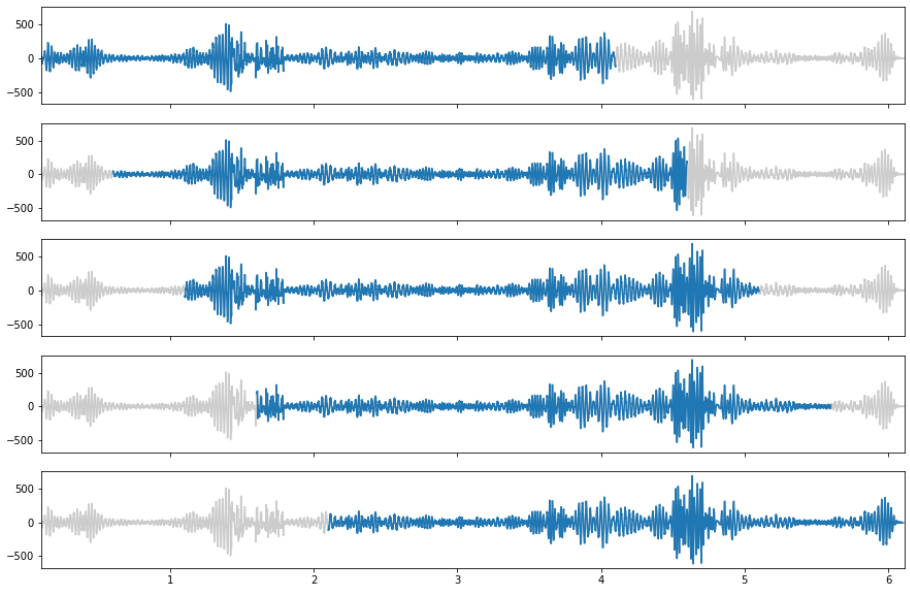


Figura 5.3: Ejemplo de la extracción de 5 frames dentro de un segmento válido. Cada porción de la señal del frame que se extrae se resalta en azul sobre la señal del segmento, en gris. La duración de cada frame es de 4 segundos, mientras que el solapamiento entre frames es de 3.5 segundos.

esta duración se asegura abarcar un patrón de firma Doppler completo para el caso más lento, que se corresponde con las muestras de tanques. A su vez, es un tiempo de observación de un blanco que puede considerarse normal en las aplicaciones radar. El solapamiento por defecto se elige en 3.5 segundos, definiendo así una porción del 87.5 % en común, luego de evaluar una relación de compromiso entre independencia y cantidad de muestras. Se preferiría definir un valor menor para tener mayor descorrelación entre frames, pero al disminuir el solapamiento se reduce la cantidad de muestras en el dataset (se extraen menos frames por segmento), como se mencionó anteriormente.

5.1.7. Generación de frames de ruido

Así como se pueden agregar al dataset muestras de fragmentos de señal que no contengan información significativa de la firma Doppler de un blanco, se pueden generar señales creadas usando modelos de ruido. De esta manera, el clasificador dispondrá de una clase **noise** que represente la ausencia de un blanco, evitando así que entregue un resultado de inferencia eligiendo una clase de blanco, por más que ésta sea improbable.

Los modelos pueden ser variados e incluirse como una sola clase. Lo conveniente sería que dichos modelos sean representativos del *clutter* al que estará expuesto el radar

en los distintos ambientes de operación. El SPL dataset original disponía de un conjunto de adquisiciones de clutter, pero no se encuentran disponibles, como se mencionó en 4.2. En la cadena de dataset se deja prevista la generación de muestras de ruido con distribuciones: *uniforme* y *gaussiana*. A continuación se muestra un ejemplo de los parámetros de configuración para la generación de muestras con ruido en la cadena:

```
{
"self.generate_noise_frames": true,
"noise_folder_path": "../data/SPL_work/noise/",
"fs_noise": 5681.81818182,
"noise_types": ["uniform", "gaussian"],
"noise_num_signals_to_gen": 20,
"power_noise": 1.0,
"bias_noise": 0.0,
"t_len_noise": 30.0,
}
```

Las señales generadas se guardan como si fueran adquisiciones de blancos y el proceso de extracción de frames es igual al que se explicó anteriormente, por lo que no hay un tratamiento especial que se realice sobre estas señales.

5.2. Mapeos

En la sección 4.4 se definió el concepto de *clase* y *mapeo*. La cadena que crea un dataset construye el mismo utilizando un único mapeo, ya que la dimensión de salida del clasificador es fija y se corresponde precisamente con el conjunto de clases sobre las que se quiere estimar las probabilidades de pertenencia de cada muestra. El framework permite construir los mapeos directamente definiendo el conjunto de *features* por clase, asignando una *etiqueta* (*label*) única a cada clase. Los mapeos que se definieron por defecto se detallan a continuación, mostrando las clases que lo componen y las *features* que conforman cada clase entre paréntesis:

- **map_1**:
  - person (person),
  - vehicle (vehicle),
  - unknown (animal).
- **map_2**:
  - one (one),
  - two (two),

- car (car),
- tank (tank),
- animal (animal).

- **map_3**:
  - one (one/normal, one/slow),
  - one-fast (one/fast, one/run),
  - two (two/normal, two/slow),
  - two-fast (two/fast, two/run),
  - car (car/slow),
  - car-fast (car/fast),
  - tank (tank),
  - confuser (animal).
- **map_4**:
  - one (one),
  - two (two),
  - car (car),
  - tank (tank),
  - animal (animal),
  - noise (noise).

Estos mapeos por defecto se definieron para evaluar el desempeño de un mismo modelo al intentar discriminar entre distintas características. El *map_1* tiene la intención de realizar una clasificación general, agrupando blancos de distintas características por clase. El *map_2* y *map_4* serían los candidatos a usarse en el clasificador a desplegar en el radar, porque presentan las clases más útiles para el usuario, pero el *map_4*, al tener el ruido (o clutter) como una clase extra, permitiría evitar falsas alarmas; convirtiéndose en el mapeo sobre el que estaremos interesados en mayor medida. El *map_3* es un mapeo mucho más exigente para el clasificador, en donde se busca evaluar el desempeño al poder discernir entre blancos del mismo tipo, pero a velocidades diferentes.

El proceso de **labeling (etiquetado)** es simplemente asignar una codificación a cada clase del mapeo que se haya seleccionado y luego asignar las etiquetas a cada muestra que conforme el dataset de acuerdo a dicho mapeo.

5.3. Partición Train/Validation/Test

En este punto de la cadena tenemos los frames extraídos de los segmentos válidos y con una clase asignada a cada uno de ellos. Se necesita realizar las particiones del

dataset: *Train*, *Validation* y *Test*.

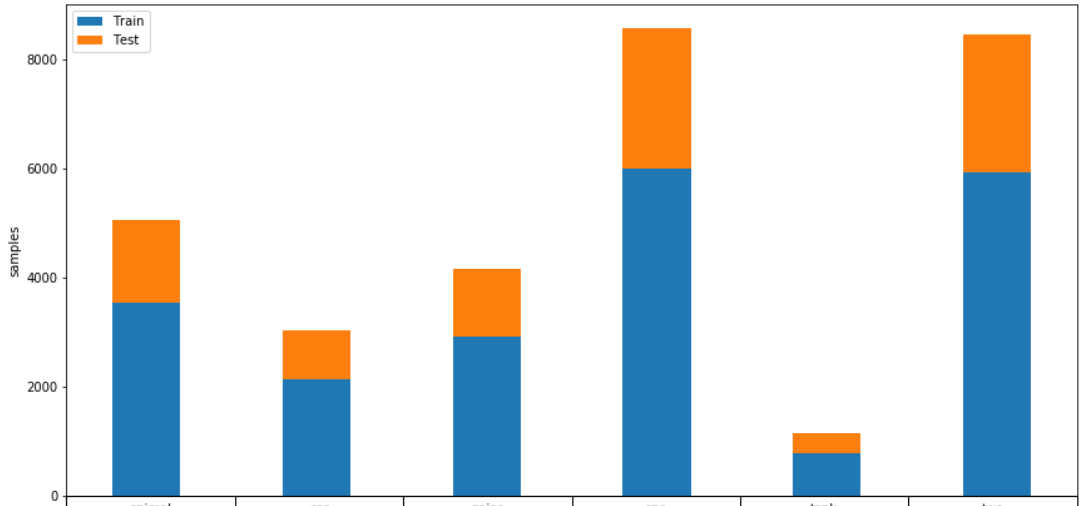
La partición **Train** tiene como objetivo alimentar al modelo del clasificador en su etapa de entrenamiento, por lo que el clasificador se especializará en clasificar dichas muestras, que se presumen como representativas de las muestras que se proporcionarán al clasificador una vez desplegado en el radar. Como es necesario evaluar la capacidad de generalización de nuestra partición de entrenamiento y de nuestro modelo, es necesario contar con un conjunto de datos independientes (que provienen de adquisiciones diferentes)[3] de la partición de entrenamiento para realizar dicha evaluación. La partición para estimar el desempeño de nuestro clasificador, con muestras con las que no fue entrenado, se denomina partición de **Test**. Como los datasets se conforman con múltiples muestras (frames) que pueden extraerse de una adquisición, esas muestras no son independientes. Es por ello que si queremos generar independencia entre las particiones de Train y Test, es necesario tener en consideración que no se pueden repartir entre ambas frames de una misma adquisición.

La partición **Validation** es necesaria cuando se realiza un ajuste de los hiper-parámetros del modelo, variando los mismos en búsqueda de un valor óptimo de las métricas de interés sobre esta partición. Se utiliza, entonces, como una partición (independiente de la de entrenamiento en la medida de lo posible) equivalente a la de *Test*, para evaluar el entrenamiento del modelo sobre datos desconocidos a medida que se modifican los hiper-parámetros. Un ejemplo sería, modificar la cantidad de unidades (neuronas) en una capa del modelo y evaluar para qué tamaño de esa capa el clasificador tuvo mejor desempeño sobre la partición *Validation*. Por más que se use una partición de validación, sigue siendo necesaria la partición de Test, ya que en una última instancia, el modelo debe ser evaluado con datos que no fueron usados para entrenarlo, ni para optimizarlo. En el desarrollo de este trabajo no se utilizó optimización de los modelos por hiper-parámetros, por lo que el tamaño de la partición de validación es cero para los diversos datasets generados. Sin embargo, si en un futuro se contara con un dataset más completo y con mayor cantidad de muestras, sería recomendable sumar esta optimización a la cadena de entrenamiento.

5.3.1. Métodos de partición

Un parámetro característico en el proceso de partición es el porcentaje de muestras que le corresponde a cada partición. En general, si el dataset es grande se utiliza 70/30 % (Train/Test), o bien, 70/10/20 % (Train/Validation/Test); pero si el dataset no es muy grande, o no se disponen de muchas muestras para algunas clases, se prefiere

[3]Cuando se hace mención a la *independencia* de los datos en el contexto de las particiones del dataset, no se refiere a la independencia estadística, sino a que provienen de adquisiciones diferentes.



	animal	car	noise	one	tank	two
Train	3540	2131	2912	6011	782	5933
Test	1516	912	1248	2575	360	2539
Total	5056	3043	4160	8586	1142	8472
Partition	0.700	0.700	0.700	0.700	0.685	0.700

Figura 5.4: Cantidad de muestras por clase y partición. Estos valores se corresponden al dataset *corona* y al *mapeo #4*, con una partición del tipo *shuffle hold-out* del 70 % para la partición de Train, usando el algoritmo *independent greedy unbalanced* para la distribución de muestras.

reducir las particiones de Validation y Test, por ejemplo, 80/20 %.

Más allá del tamaño de las particiones, en algunos casos es necesario elegir cuidadosamente el método con el que se particionarán los datos. Lo más común es que las particiones se realicen mezclando aleatoriamente todas las muestras y luego quedarse con las primeras N_{train} muestras para Train, luego las N_{val} muestras para Validation y por último las N_{test} muestras para Test. Este método de partición suele denominarse **Shuffle Hold-Out** [63]. Este método es sencillo, pero tiene la desventaja de que puede generar datasets desbalanceados (cantidad distinta de muestras de entrenamiento por clase) si es que el dataset completo no tiene una cantidad balanceada de muestras por clase. Tampoco, en esta aplicación, aseguraría la independencia de muestras entre las particiones.

Existe otro método que se denomina **k-fold cross-validation**[63], en donde al dataset completo se lo divide en k particiones y se utiliza una de ellas para Test y el resto para entrenamiento para un ciclo particular. En el próximo ciclo, se elige una partición distinta para Test y el resto para entrenamiento; y así sucesivamente hasta completar k ciclos. De esa manera, en cada ciclo de entrenamiento se utilizó cada muestra una sola vez para evaluar el desempeño. La métrica por época de entrenamiento suele ser un promedio de las métricas obtenidas por cada ciclo. Este método es útil cuando no se dispone de un dataset grande, pero tampoco asegura independencia de las muestras, y es por eso que no se utiliza en este desarrollo.

En la sección 4.5 se mostró que la cantidad y duración de las adquisiciones era muy

diferente por cada clase, por lo que se presenta el caso de un dataset desbalanceado, lo que puede generar sesgos en las inferencias y dificultades en la partición del dataset. Una opción podría ser quedarse con la cantidad de muestras para entrenamiento igual a la de la clase que tiene menos, pero eso nos generaría un dataset de entrenamiento muy chico, desperdiciando una gran cantidad de muestras. Finalmente se implementa un método que se denominó **independent greedy unbalanced**, en donde se crean grupos de muestras independientes (agrupadas por adquisición) y separados por clase; luego se busca la mejor distribución de esos grupos entre las particiones para lograr la proporción buscada entre las cantidades de muestras por partición. [64]

En la figura 5.4 puede verse un ejemplo de la aplicación de este algoritmo usando el *mapeo #4*. Notar que la proporción de datos entre Train y Test es 70 % en casi todas las clases, salvo para *tank* en donde se aproxima bastante (esto se debe a que con menos muestras es más difícil aproximar las proporciones manteniendo las independencia entre particiones). Claramente, se tiene una condición de desbalance de las muestras de entrenamiento entre clases, pero esto se solucionará con el proceso de *Data-Augmentation* que se describe en la siguiente sección.

Los parámetros de configuración a definir se detallan a continuación. Si no se especifican las proporciones para Validation y Test se deben considerar como 0 % para Validation y el resto para Test. El parámetro *seed* especifica la semilla para realizar la permutación aleatoria (shuffle) de las muestras de cada partición; si se define como *None* el resultado de la permutación será distinto cada vez que se ejecute la cadena:

```
{
"train_size": 0.7,
"seed": 0
}
```

5.4. Data-augmentation

La etapa que sigue en la *Dataset Chain* es la de *Data Augmentation*. En la sección 3.2 se explicó que sobre la partición *Train* se generan muestras nuevas a partir de las disponibles hasta el momento en esta partición, buscando mejorar el desempeño del clasificador. Con el incremento en la cantidad de muestras se busca mejorar el desempeño general del clasificador, principalmente mejorando la capacidad de generalización del mismo. De la misma forma, este incremento en la cantidad de muestras debería acelerar la convergencia del entrenamiento. Con esta etapa, se buscará adicionalmente ecualizar la cantidad de muestras por clase, ya que es posible discriminar la cantidad de nuevas muestras que se generarán por cada clase.

Debido a la naturaleza de las señales que se quieren clasificar, las distorsiones que se introducen a las muestras disponibles son:

- **time contraction** (contracción temporal): Se realiza un downsampling de la señal Doppler temporal para simular que los eventos temporales sucedieron más rápido.
- **time dilation** (dilatación temporal): Se realiza un upsampling de la señal Doppler para simular que los eventos temporales sucedieron más lento.
- **noise addition** (adición de ruido): Se agrega ruido a la señal Doppler temporal.

El parámetro de la Dataset Chain que define los tipos de métodos a usar para generar nuevas muestras, es una lista que se muestra a continuación:

```
{
 "augmentation_type": ["time_dilation", "time_contraction", "noise_addition"]
}
```

5.4.1. Re-muestreo

La generación de nuevas muestras a partir de *time contraction* y *time dilation* se basan en el hecho de que los fenómenos observados a través de la firma Doppler son igualmente válidos si a las componentes espectrales se le aplica una escala en el dominio de la frecuencia. Por ejemplo, una persona caminando o corriendo a mayor velocidad, mostrará componentes espectrales a mayor frecuencia como puede verse en la ecuación 2.14. Entonces, si tenemos una adquisición de una persona caminando a una velocidad v determinada, se podrán generar nuevas muestras útiles de esa misma persona a distintas velocidades (dentro de un rango adecuado) aplicando un factor de escala s_f en el dominio de la frecuencia, como se muestra en la ecuación 5.13.

$$f'_D = \frac{2v'}{\lambda_{Tx}} = \frac{2 \cdot s_f \cdot v}{\lambda_{Tx}} = s_f \cdot f_D \quad ; |s_f| > 0 \tag{5.13}$$

Notar que este cambio en el dominio de la frecuencia modificará la señal en el dominio del tiempo. Esto se puede ver a través de la propiedad que vincula el cambio de escala a través de la Transformada de Fourier, mostrada en la ecuación 5.14.

$$\mathcal{F}\{\boldsymbol{x}(rt)\} = \frac{1}{|r|}\boldsymbol{X}\left(\frac{f}{r}\right) \quad ; |r| > 0 \tag{5.14}$$

Como las señales se encuentran muestreadas, para generar esta modificación de escala en estas señales, se debe realizar un **resampling** (re-muestreo). El factor de

cambio de escala r, se denomina *resampling factor (factor de re-muestreo)*, y podría ser cualquier número racional expresándose $r = r_u/r_d$. El proceso de re-muestreo se realiza haciendo un *upsampling (interpolación)* por un factor r_u y luego un *downsampling (decimación)* por un factor r_d. De esa manera, las componentes Doppler se modificarán en función del resampling factor que se elija, según 5.15. Notar que si r es negativo, se genera una inversión temporal de la señal, lo que podría resultar igualmente útil, pero que no se utilizó en el desarrollo de los datasets. En la figura 5.5 se muestran los espectrogramas de un frame de una señal original, y luego los espectrogramas de frames extraídos del re-muestreo del segmento original, mostrando los casos de factores de re-muestreo de 0,9 y 0,5; comprobándose cómo la escala en tiempo modifica de manera inversa la escala en frecuencia.

$$\boldsymbol{x}'(t) = \boldsymbol{x}(rt) \rightarrow f'_D = \frac{f_D}{r} \quad ; |r| > 0 \tag{5.15}$$

En la *Dataset Chain* se generarán nuevas señales haciendo un re-muestreo de todas las adquisiciones usando un conjunto de valores como factores de re-muestreo; o sea, de cada adquisición se generarán tantas "adquisiciones"nuevas como factores de re-muestreo se elijan. Los parámetros que definen el rango en el que variarán los factores de re-muestreo para los casos de *time contraction* y *time dilation* son los siguientes:

```
{
"max_contraction": 0.5,
"min_contraction": 0.9,
"max_dilation": 2.0,
"min_dilation": 1.1
}
```

5.4.2. Adición de ruido

El método restante para aumentar la cantidad de datos es agregar ruido a las señales originales, a este proceso se lo denomina *noise addition (adición de ruido)*. En este caso, partiendo de cada uno de los segmentos válidos del dataset de entrenamiento se agrega **ruido gaussiano** para generar un nuevo segmento, si bien podrían usarse distintos modelos de ruido, por ahora se acota la selección a este tipo de ruido como modelo de ruido térmico.

La parametrización de esta adición de ruido se realiza mediante el valor de SNR que se desea en la señal resultante. De esta manera, se define un rango de valores de SNR, para generar una nueva señal por cada valor de SNR que se elija dentro de ese rango. Dado un valor de SNR, se calcula la potencia máxima de la señal original para poder

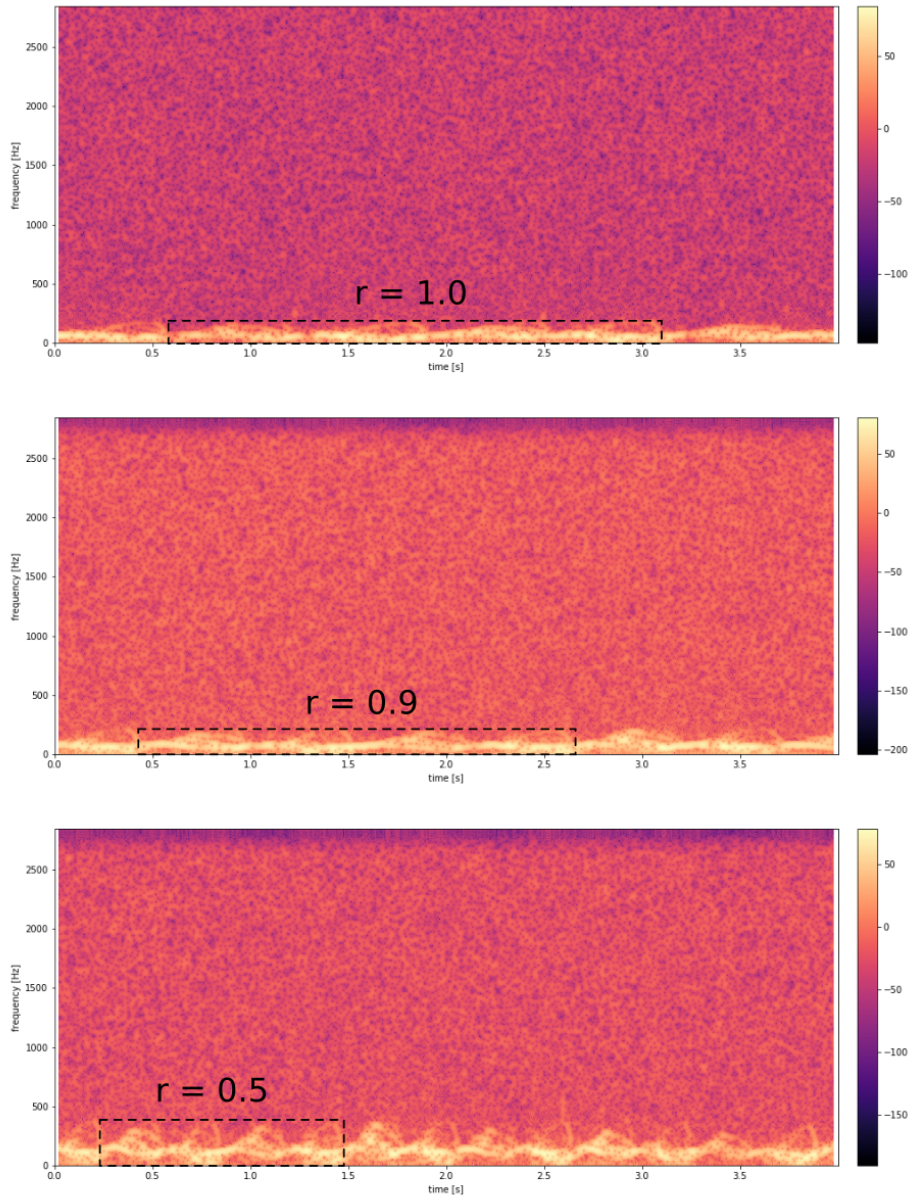


Figura 5.5: Espectrogramas de muestras obtendidas usando data-augmentation por re-muestreo. Se muestra un frame de un segmento válido (arriba, señal original) al que no se le ha aplicado re-muestreo ($r = 1{,}0$). En el medio se muestra el frame (con igual tiempo de inicio) de una nueva señal generada a partir del segmento anterior, aplicando re-muestreo con $r = 0{,}9$) Abajo se muestra el frame correspondiente con re-muestreo de $r = 0{,}5$. En todas las figuras se resaltan dos ciclos de la señal Doppler del blanco, considerando la variación máxima de frecuencia Doppler. Notar que para $r = 0{,}5$ se duplica la frecuencia máxima y se reduce a la mitad el tiempo de ciclo.

calcular la potencia de ruido necesaria para alcanzar el valor de SNR. La componente de ruido es una realización aleatoria en el dominio temporal, de media cero y varianza que verifica lo anterior (ver 5.16, que luego es sumada a la señal temporal original para obtener el nuevo segmento.

$$\mathrm{Var}(S_{\mathrm{noise}}) = \sigma^2 = 10^{(\mathrm{máx}\{\boldsymbol{S}_{dB}\} - \mathrm{SNR}_{dB})/20} \tag{5.16}$$

Entendiéndose a máx$\{\boldsymbol{S}_{dB}\}$ como la potencia máxima de la señal que se obtuvo luego de calcular el espectrograma de la misma, en decibeles, y a σ como la desviación de la distribución normal que da origen a la realización del ruido en este caso.

Los parámetros de la Dataset Chain que definen el rango de SNR con que se generarán las nuevas señales con ruido agregado son:

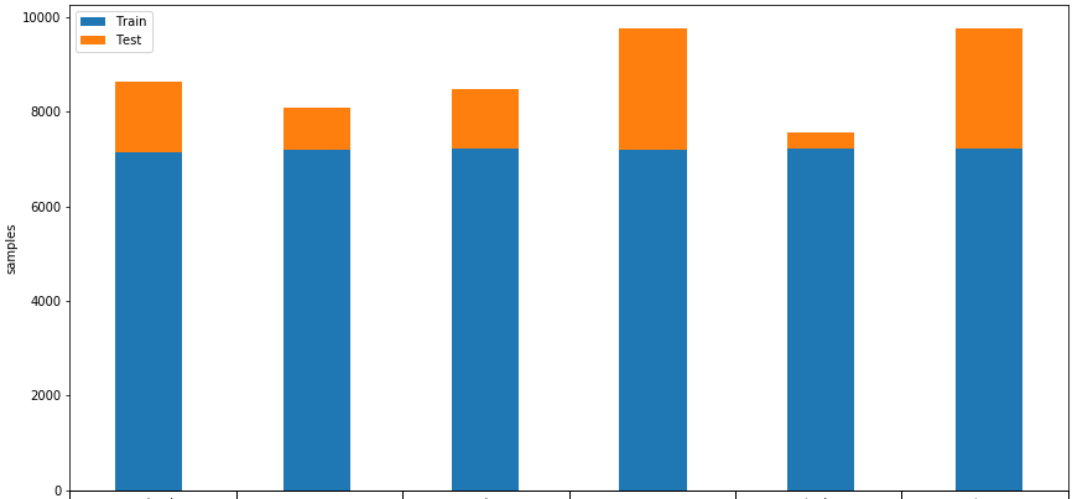
```
{
"SNR_max": 50.0,
"SNR_min": 20.0
}
```

5.4.3. Cantidad de nuevas muestras

La cantidad de muestras nuevas que se generarán en esta etapa dependen de muchos factores. El primero es un parámetro general que indica el porcentaje de muestras nuevas que se quieren generar para la clase que más muestras dispone. Dicho de otra manera, es el factor que definirá la cantidad de muestras máxima que tendrá una (o cualquier) clase luego de este proceso. Este factor se denomina *mult_factor_of_max*. Una vez definido este valor es necesario definir en qué porcentaje se generarán las nuevas muestras entre los distintos métodos. Para ello están los parámetros *ratio_time_contraction*, *ratio_time_dilation* y *ratio_noise_addition* que definen el porcentaje de muestras a generar por método (la suma de los tres debería ser siempre igual a uno).

Esta etapa, además de colaborar a mejorar la capacidad de generalización del modelo, se utilizará para ecualizar la cantidad de muestras por clase ya que la cantidad de muestras a generar por clase se ajusta para igualar a la cantidad final de muestras de la clase que más tiene; lo que colabora en la disminución de sesgos en las inferencias que realice el clasificador. En la figura 5.6 se muestra el resultado de aplicar Data Augmentation al dataset mostrado en la figura 5.4. Notar que se ha balanceado la partición de entrenamiento y no se ha modificado la partición de test.

En este punto ya se dispone de la cantidad de muestras (frames) que tiene cada clase, la cantidad total que se debe generar para esta misma clase y la cantidad que se deben generar por cada método, que llamaremos $M'_{f,\text{method}}$. Lo que aún no está definido es qué valores se usarán para parametrizar cada método, por ejemplo los valores para el factor de re-muestreo dentro del rango que se especificó. Como se mencionó anteriormente, cada segmento nuevo que genera un método tiene asociado un único valor de parametrización diferente. Necesitamos calcular, entonces, la cantidad de segmentos nuevos a generar, o bien, la cantidad de valores de parametrización distintos, dicha cantidad la denominaremos M_p. Por simplicidad, los valores de parametrización se elegirán igualmente distribuidos dentro del rango especificado para cada método.



	animal	car	noise	one	tank	two
Train	7129	7182	7224	7189	7210	7210
Test	1516	912	1248	2575	360	2539
Total	8645	8094	8472	9764	7570	9749
Partition	0.825	0.887	0.853	0.736	0.952	0.740

Figura 5.6: Cantidad de muestras por clase y partición luego de la etapa de Data Augmentation. Estos valores se corresponden al dataset *corona* y al *mapeo #4*, aplicando los métodos de *time dilation*, *time contraction* y *noise addition* para el incremento de los datos del dataset de entrenamiento.

Por ejemplo, si para *time dilation* se definió un rango para el factor de re-muestreo de 1,1 a 2,0, si $M_p = 10$, se generarán 10 segmentos nuevos con los factores de re-muestreo: 1,1; 1,2; ...; 1,9; 2,0.

Si la duración de cada frame (*frame_len*) y el solapamiento entre frames (*frame_overlap*) está definido, entonces la cantidad de frames nuevos que se puede extraer de un segmento generado viene dado por la ecuación 5.12. Para generar $M'_{f,\text{method}}$ se tendrán que generar M_p nuevos segmentos según:

$$M_p = \left\lceil \frac{M'_f}{M_f} \right\rceil = \left\lceil \frac{M'_f}{\left\lfloor \frac{N'_s - N_{fov}}{N_f - N_{fov}} \right\rfloor} \right\rceil \tag{5.17}$$

Notar que la longitud del segmento se expresa como N'_s, ya que la duración del segmento puede depender del método y su parametrización. En el caso en donde los métodos modifican la duración del segmento generado, como *time dilation* y *time contraction*, se puede escribir N'_s como la duración media de los segmentos generados, dado un segmento original, un método y el conjunto de valores de parametrización. Entonces:

$$N'_s(\{r_i\}) = \overline{N_s(\{r_i\})} = N_s \cdot \overline{r} \tag{5.18}$$

Entiéndase $\{r_i\}$ como el conjunto de valores que usa el método como parámetro para generar cada segmento nuevo.

En resumen, para especificar la cantidad de muestras a generar es importante conocer la cantidad de muestras de la clase más voluminosa, para así poder especificar el multiplicador general. La duración y solapamiento de frames especificados definirá la variedad de nuevos segmentos a generar para poder alcanzar las cantidades a generar. Entonces, los parámetros de configuración restantes para esta etapa son:

```
{
"frame_len": 4.0,
"frame_overlap": 1.5,
"mult_factor_of_max": 1.2,
"ratio_time_contraction": 0.4,
"ratio_time_dilation": 0.4,
"ratio_noise_addition": 0.2,
}
```

Una alternativa a definir el solapamiento entre frames puede ser definir cuántos segmentos nuevos deben generarse por cada segmento original, o sea, forzar el valor de M_p. En este caso, como la cantidad de frames nuevos sigue siendo un valor conocido, hay que calcular el valor de solapamiento entre frames para poder extraer la cantidad de frames suficientes por segmento. Entonces:

$$M_f = \left\lceil \frac{M'_f}{M_p} \right\rceil \tag{5.19}$$

Partiendo de 5.12, se puede obtener la cantidad de muestras a solapar dependiendo de la duración del nuevo segmento:

$$N_{ov} = N_f - \left\lceil \frac{N'_s - N_f}{M_f - 1} \right\rceil \tag{5.20}$$

5.5. Pre-Procesamiento de las muestras

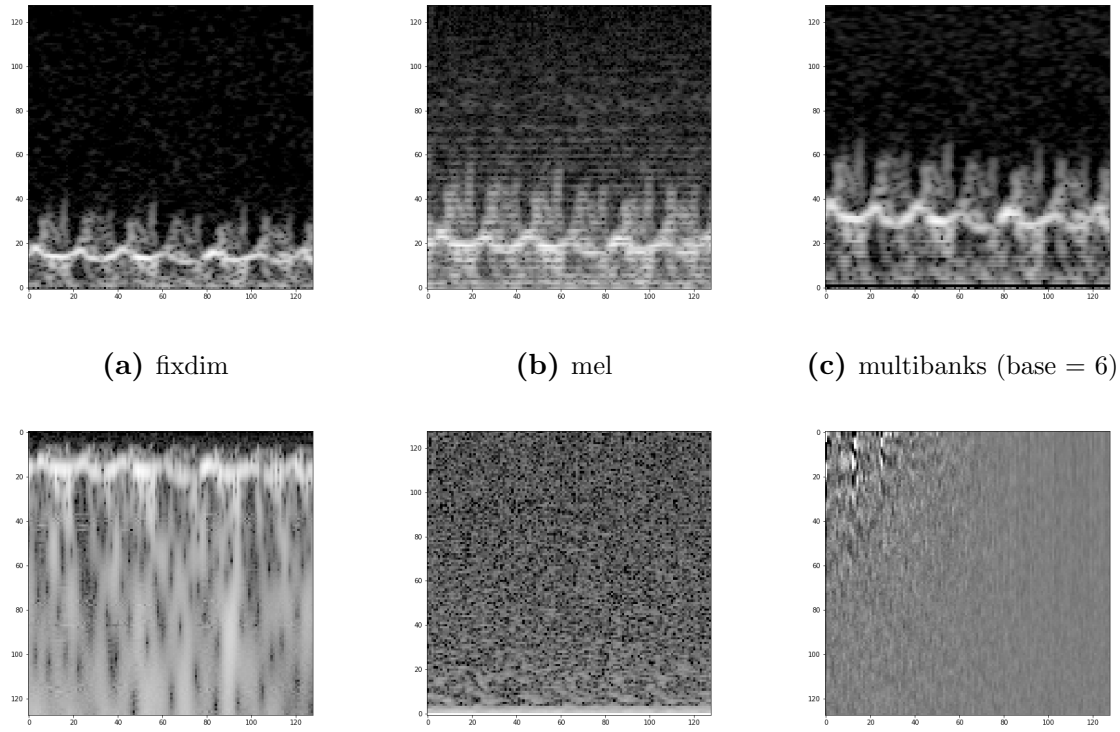
En este punto de la Dataset Chain ya se diponen de todas las muestras temporales (frames) en las distintas particiones del dataset. Como los modelos de clasificador para este desarrollo toman imágenes como entrada, es necesario convertir cada frame (señal temporal) a una imagen de tamaño fijo, ya que la entrada del clasificador no puede tomar datos de dimensiones distintas. Esta conversión a imágenes es la tarea principal de la etapa de *Pre-Processing (pre-procesamiento)*, pero pueden existir algunos procesamientos adicionales que se incluyen en esta etapa antes de almacenar las muestras definitivas del dataset. Esta etapa será compartida por la Dataset Chain (ver 3.2), como por la cadena de clasificación (ver 3.1).

Primero se aplica una decimación para trabajar con la porción del espectro que contiene información útil, de acuerdo a cómo se distribuyen las componentes de frecuencia de las distintas firmas Doppler. En la sección 4.5.2 se hizo la observación de que la frecuencia máxima Doppler de las muestras rondaba los 800 $[Hz]$ (valor conservador, puesto que la mayoría de las muestras presenta componente por debajo de los 500 $[Hz]$); entonces, como la frecuencia de muestreo es $f_s = 5681{,}8\ [Hz]$. Se elige por defecto usar un factor de decimación de 4 para todas las muestras, por lo que la frecuencia Doppler máxima será de 710,2 $[Hz]$ (a partir de calcular $(f_s/2)/4$), que según la ecuación 2.14, equivale a una velocidad radial (máxima no ambigua) de 42,6 $[km/h]$ (también se considera aceptable elevar la decimación hasta un factor de 8). Si no se aplicara un filtro anti-aliasing, las componentes Doppler correspondientes a velocidades mayores (que se corresponderían sólo con automóviles en este dataset) presentarán un plegado; que no debería afectar significativamente el desempeño del clasificador en cuanto el espectro plegado sea similar al de velocidades menores, o bien, en cuanto se dispongan de muestras con plegado para el entrenamiento del clasificador. Se podría, alternativamente, aplicar un factor de decimación mayor para las clases con velocidades menores, como personas o algunos animales, pero eso complejizaría la cadena de clasificación cuando, a priori, no se conoce el tipo de blanco; obligando a decimar con todos los valores usados durante el entrenamiento. La decimación se realiza de manera convencional, aplicando un filtro anti-aliasing, por lo que no se describirá en detalle esta operación.

Como segundo paso, se debe convertir cada frame (en el formato señal temporal) a una imagen de formato predefinido. Luego de evaluar distintas configuraciones, se decidió definir un tamaño de imagen de 128×128 píxeles, con un sólo canal de color (escala de grises), y con 8 bits por píxel en el formato *UINT8* (entero sin signo de 8 bits). Resta entonces, definir el método con que se convertirá cada señal en una imagen. Se evaluaron diversos métodos, en donde sus parámetros se ajustan para conseguir como salida la imagen en el formato adecuado, estos métodos se listan a continuación con su identificador entre paréntesis:

- Fixed Dimensions Spectrogram (*fixdim*)
- Mel Spectrogram (*mel*)
- Multibanks Spectrogram (*multibanks*)
- Fixed Dimensions Scalogram (*scalogram*)
- Mel Frequency Cepstral Coefficients (*mfcc*)
- 2D Cosine Transformed Spectrogram (*dct*)

Cada método entrega una imagen cuyos valores deben ajustarse al rango impuesto por el tipo de dato elegido para los píxeles de la imagen final. Esta conversión tiene en



(a) fixdim **(b)** mel **(c)** multibanks (base = 6)

(d) scalogram (complex morlet) **(e)** mfcc **(f)** dct

Figura 5.7: Métodos conversión a imágenes disponibles en la Dataset Chain. Las imágenes que se muestran se corresponden a una misma muestra (una persona caminando) convertida a través de los distintos métodos, bajo parámetros específicos. En todos los casos se usa una decimación de 4. (a) usa el espectrograma convencional (window = hamming, wlen = 256, dynamic range = 100 dB). (b) espectrograma en escala Mel (window = hamming, wlen = 256, dynamic range = 70 dB, fs_mult = 4). (c) usa mapeo a banco de filtros (base = 6, window = hamming, wlen = 320, dynamic range = 100 dB). (d) usa el escalograma obtenido con CWT (wavelet = cmor1-1.5 (complex morlet wavelet), level = 200, dynamic range = 60 dB). (e) usa el método MFCC (window = hamming, wlen = 256, dynamic range = 70 dB). (f) usa la DCT 2D del espectrograma (window = hamming, wlen = 256, dynamic range = 100 dB, scale = linear).

cuenta el rango dinámico de los valores obtenidos muestra a muestra y se detalla en 5.5.7.

En la figura 5.7 se muestran las imágenes resultantes al aplicar los distintos métodos (usando parámetros específicos) a un mismo frame. A grandes rasgos, el método *fixdim* es el método convencional y que demanda menos procesamiento; los métodos mel y multibanks buscan ampliar la resolución en frecuencia para las frecuencias Doppler bajas, donde se concentra la información para las clases como personas o animales; el método *scalogram* (usando wavelets) busca mejorar el problema de la relación temporal-frecuencial constante que presenta la STFT, ayudando a resolver mejor las frecuencias Doppler bajas; por último, los métodos *mfcc* y *dct* se basan en métodos de compresión de información, permitiendo la clasificación en función de una transformación adicional. En las secciones siguientes se presentará una breve descripción de los distintos métodos

de conversión y los parámetros particulares de cada uno de ellos.

Los parámetros generales de la etapa de pre-procesamiento de la Dataset Chain son entonces los que definen la decimación, el método de conversión de señal temporal a imagen, el tamaño de la imagen y el tipo de dato por pixel. Un ejemplo de estos parámetros en el archivo de configuración se muestra a continuación:

```
{
"preprocessing_type": "fixdim",
"output_dim": [128,128],
"dtype": "uint8",
"decimation": 4
}
```

5.5.1. Fixed Dimensions Spectrogram

En este método se calcula el espectrograma convencional, explicado en 5.1 para convertirlo en una imagen. Como no se desea hacer una interpolación del espectrograma para convertilo en una imagen del tamaño deseado, ya que sería un proceso que demanda mayor procesamiento, lo conveniente es ajustar los parámetros del espectrograma para que sus dimensiones resultantes coincidan con las dimensiones requeridas para la imagen.

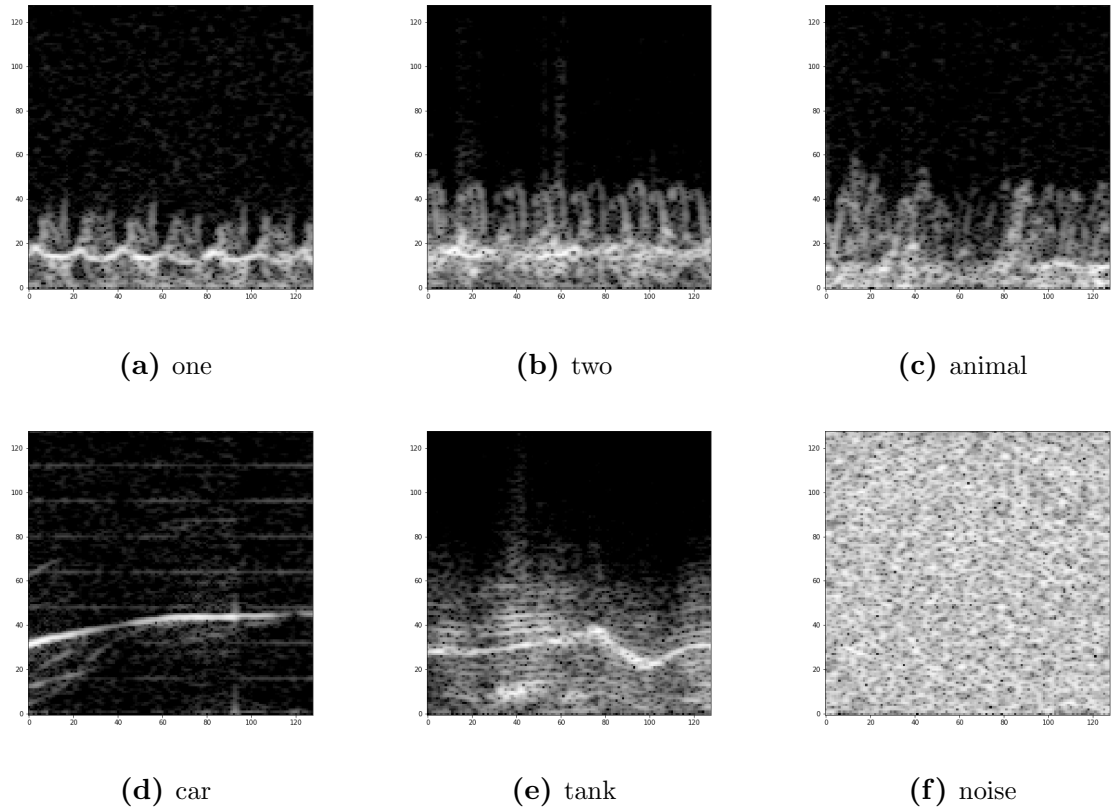
En la ecuación 5.1 se puede observar que la dimensión temporal N_c de la STFT, por lo tanto del espectrograma, depende de la cantidad de muestras de la señal $N_{\boldsymbol{x}}$ (fija ahora puesto que se trata de un frame), de la longitud de la ventana W_{len} que será un parámetro a definir y de la cantidad de muestras a solapar N_{ov} que será nuestro parámetro a ajustar para que $N_c = N_{pxt}$ (cantidad de píxeles en la dimensión temporal, o bien horizontal). Entonces:

$$N_{ov} = W_{len} - \left\lfloor \frac{N_{\boldsymbol{x}} - W_{len}}{N_{pxt} - 1} \right\rfloor \tag{5.21}$$

Notar que el redondeo se hizo para abajo, para asegurar un N_{ov} mayor, asegurando que $N_c \geq N_{pxt}$, y de esa forma poder recortar el espectrograma a N_{pxt} píxeles en la dimensión temporal.

La cantidad de puntos en la dimensión de las frecuencias, como se explicó en 5.1, viene dada por el tamaño de la FFT a realizar N_{FFT}. Con el fin de optimizar el cálculo de cada DFT a través de la FFT, se elige que la cantidad de puntos para la FFT sea la potencia de 2 mayor más cercana y se usará zero-padding de ser necesario. Entonces:

$$N_{FFT} = 2^{\left\lceil \log_2(2\,N_{pxf}-1) \right\rceil} \tag{5.22}$$



(a) one **(b)** two **(c)** animal

(d) car **(e)** tank **(f)** noise

Figura 5.8: Conversión a imágenes usando el método *fixdim* (espectrograma con dimensiones fijas). Se muestra la conversión de muestras (frames) correspondientes a las clases del mapeo #4. La configuración utilizada para este método fue: window = hamming, wlen = 256, dynamic range = 100 dB

Donde N_{pxf} es la cantidad de píxeles en la dimensión de las frecuencias. Tener en cuenta que, como las señales disponibles son reales, sólo es necesaria una mitad del espectrograma (ya que será simétrico respecto de las frecuencias). Por ello la cantidad de puntos en el eje de las frecuencias será $N_{FFT}/2+1$ (se incluye el valor de frecuencia cero). En el caso por defecto, en donde se requieren 128 píxeles para la dimensión de las frecuencias, tenemos $N_{FFT} = 256$. Una restricción a cumplir es que $W_{len} \leq N_{FFT}$.

Este método es seleccionado por defecto en la cadena de procesamiento, debido a que ha mostrado buenos resultados y el cómputo demandado es el menor en comparación a los otros métodos. En la figura 5.8 se muestran los resultados de convertir una muestra de cada clase (usando el mapeo # 4).

El identificador de este método es *fixdim* y los parámetros de configuración correspondientes se muestran en el siguiente ejemplo:

```
{
"preprocessing_type": "fixdim",
"window": "hamming",
"wlen": 256,
```

```
"dynamic_range_dB": [100]
}
```

5.5.2. Mel Spectrogram

Este método se basa en mapear las frecuencias obtenidas en el espectrograma a la *escala de Mel*. Esta escala está relacionada con la percepción humana de las diferencias frecuenciales de los sonidos, donde a frecuencias bajas el oído humano puede percibir mejor diferencias chicas entre frecuencias, mientras que para altas frecuencias es más difícil percibir iguales diferencias. El mapeo se realiza usando ventanas triangulares solapadas que se encuentran más juntas a bajas frecuencias (mejor resolución), y viceversa para altas frecuencias, donde la cantidad de filtros se definirá igual a la cantidad de píxeles que se requieren en el eje de las frecuencias. Por lo general, la transformación también modifica la amplitud en las distintas bandas emulando la sensibilidad del oído humano. El mapeo de frecuencias (centros de los filtros), entonces, se realiza a través de una función no lineal; donde la ecuación que generalmente se utiliza es:

$$m = 2595 \log_{10}\left(1 + \frac{f}{700}\right) \tag{5.23}$$

Lo que se persigue con este método es mejorar la resolución en las señales Doppler de persona o animales, que presentan la mayoría de sus componentes en bajas frecuencias. Sin embargo, la escala de Mel presenta una zona quasi-lineal hasta frecuencias mayores a $1000\,Hz$ por lo que, para las clases mencionadas, no se logra una mejora considerable de la resolución. Como la conversión tiene en cuenta la frecuencia de muestreo, se puede cambiar la escala de la transformación aplicando un factor que denominaremos *fs_mult*, de manera que podemos ampliar la resolución en las bajas frecuencias usando valores mayores a 1 para este multiplicador. Un ejemplo del resultado de este tipo de conversión puede verse en 5.7b.

El identificador de este método es *mel* y los parámetros de configuración correspondientes se muestran en el siguiente ejemplo:

```
{
"preprocessing_type": "mel",
"window": "hamming",
"wlen": 256,
"dynamic_range_dB": [70],
"fs_mult": 4
}
```

5.5.3. Multibanks Spectrogram

Este método utiliza, al igual que el método Mel Spectrogam, el mapeo de frecuencias a través de función logarítmica que define los centros de frecuencia de filtros triangulares. La transición entre filtros es lineal asegurando que se mantenga conservada la potencia de la señal. Se busca obtener mayor resolución en las bajas frecuencias a costa de perder resolución en las altas frecuencias. En este caso, la función de mapeo no se corresponde con el comportamiento del oído humano, utilizándose:

$$f_o = (N_o - 1)\log_{\mathrm{b}}\left(1 + \frac{f_i(b-1)}{(N_i - 1)}\right) \qquad f_i = 0, ..., N_i - 1 \tag{5.24}$$

Donde N_i es la cantidad de puntos de la frecuencia de entrada (original) y N_o la cantidad de pulsos de la frecuencia de salida (mapeada). Entonces, f_i y f_o son los valores de frecuencia normalizada de entrada y mapeada correspondientemente. El parámetro b se denomina **base** y determina la base logarítmica (o exponencial si se mira de manera inversa el mapeo).

En el caso en que el espectrograma generado previamente al mapeo ya tiene la dimensión de frecuencias correcta, $N_o = N_i = N_{pxf}$; de otra forma $N_i = N_{FFT}/2 + 1$ (para señales reales) o $N_i = N_{FFT}$ (para señales complejas), y $N_o = N_{pxf}$. El parámetro base se ajustará en función de cuánta más resolución queramos en las bajas frecuencias a costa de la resolución en las altas frecuencias. Cuanto mayor sea el valor de la base más desbalance habrá entre dichas resoluciones. Hay que tener presente que, cuanto mayor sea la resolución en las bajas frecuencia, mayor cantidad de filtros se repartirán las porciones entregadas por la FFT, por lo que pueden aparecer filtros con poca energía, dando lugar a artefactos en la imagen que se manifiestan como líneas horizontales oscuras (baja densidad de potencia espectral). Esto se puede prevenir, o atenuar, incrementando la cantidad de puntos de la FFT (en la versión actual de la implementación $W_{len} = N_{FFT}$, por lo que incrementando el tamaño de la ventana se incrementaría la cantidad de puntos de la FFT).

El identificador de este método es *multibanks* y los parámetros de configuración correspondientes se muestran en el siguiente ejemplo:

```
{
"preprocessing_type": "multibanks",
"window": "hamming",
"wlen": 320,
"dynamic_range_dB": [100],
"base": 6
}
```

En la figura 5.7c se muestra un ejemplo del resultado de este método. Notar, comparando con 5.7a, cómo las bajas frecuencias ocupan una mayor porción de la imagen. También pueden observarse algunos artefactos, como líneas horizontales, pero que son menores que los obtenidos en el método *mel* (figura 5.7b); sin embargo, si se acorta la longitud de la ventana o se eleva el valor de la base, los artefactos aparecen de manera más significativa.

5.5.4. Fixed Dimensions Scalogram

Una alternativa a utilizar la STFT es utilizar la Continous Wavelet Transform (Transformada de Ondita Continua) (CWT), [65, 66]. En este método se elige una función del tipo wavelet a multiplicar con la señal. Se realiza un determinado número de productos modificando la función wavelet mediante un factor de escala y un factor de desplazamiento. La CWT de la señal $\boldsymbol{s}(t)$ puede expresarse como:

$$\boldsymbol{S}_w(a,b) = \frac{1}{|a|^{1/2}} \int_{-\infty}^{\infty} \boldsymbol{s}(t)\psi^*\left(\frac{t-b}{a}\right)dt \tag{5.25}$$

Donde $\psi(t)$ es la función *wavelet madre*, y se elige en función del fenómeno que se quiere observar a través de la transformada. El super-índice $*$ indica complejo-conjugado. Esta función se modifica a través de un *factor de escala* $a \in \mathbb{R}^{+*}$ que contrae o alarga la función wavelet original, y es éste el que permite observar comportamientos a distintas frecuencias o alteraciones temporales de la señal. El factor b se denomina *factor de transición* o de *desplazamiento* y permite aplicar la función a distintos instantes de tiempo. Creando un conjunto de valores (a, b) se construye el **escalograma**, donde una dimensión es el valor de escala aplicado (se relaciona con la frecuencia Doppler) y la otra dimensión es el desplazamiento (se relaciona con la dimensión temporal de la señal). La cantidad de valores de a y b se deberá corresponder con la dimensión de la imagen (escalograma) resultante, por lo tanto, con los valores N_{pxf} y N_{pxt} correspondientemente. En la figura 5.7d se muestra un ejemplo del escalograma obtenido a partir de la señal de un frame.

La parametrización de este método es el tipo de *wavelet* (parámetro homónimo) que se quiere usar en la transformada y los valores de escala (por defecto se definen como valores enteros continuos desde 1 hasta un máximo a especificar con el parámetro *level*). El identificador de este método es *scalogram*. Un ejemplo de los parámetros de configuración se muestra en el siguiente ejemplo:

```
{
"preprocessing_type": "scalogram",
"wavelet": "morl",
```

```
"level": [60],
}
```

La funciones wavelet soportadas son las disponibles en la librería *pywavelets*, correspondientes al módulo *cwt*. Las funciones disponibles al momento de escribir el documento son:

Mexican Hat : Su identificador es *mexh*.

$$\psi(t) = \frac{2}{\sqrt{3}\sqrt[4]{\pi}} e^{-\frac{t^2}{2}} \left(1 - t^2\right) \tag{5.26}$$

Morlet : Su identificador es *morl*.

$$\psi(t) = e^{-\frac{t^2}{2}} \cos(5t) \tag{5.27}$$

Complex Morlet : Su identificador es *cmorB-C*. Donde B es el ancho de banda y C la frecuencia central. Ejemplo: *cmorl1-1.5*.

$$\psi(t) = \frac{1}{\sqrt{\pi B}} e^{-\frac{t^2}{B}} e^{\mathrm{j}2\pi Ct} \tag{5.28}$$

Gaussian Derivative : Su identificador es *gausP*. Donde P es un entero entre 1 y 8, que se corresponde con la P-ésima derivada de la función. En la ecuación C es un factor de normalización que depende del órden de la derivada. Ejemplo: *gaus2*.

$$\psi(t) = Ce^{-t^2} \tag{5.29}$$

Complex Gaussian Derivative : Su identificador es *cgausP*. Donde P es un entero entre 1 y 8, que se corresponde con la P-ésima derivada de la función. En la ecuación C es un factor de normalización que depende del órden de la derivada. Ejemplo: *cgaus4*.

$$\psi(t) = Ce^{-\mathrm{j}t}e^{-t^2} \tag{5.30}$$

Shannon : Su identificador es *shanB-C*. Donde B es el ancho de banda y C la frecuencia central. Ejemplo: *shan0.2-0.9*.

$$\psi(t) = \sqrt{B}\frac{\sin(\pi Bt)}{\pi Bt} e^{\mathrm{j}2\pi Ct} \tag{5.31}$$

Frequency B-Spline : Su identificador es *fpspM-B-C*. Donde M es un valor entero que representa el orden spline, B es el ancho de banda y C la frecuencia central. Ejemplo: *fbsp2-0.2-1.0*.

$$\psi(t) = \sqrt{B}\left[\frac{\sin(\pi B\frac{t}{M})}{\pi B\frac{t}{M}}\right]^{M} e^{2\mathrm{j}\pi Ct} \tag{5.32}$$

5.5.5. Mel Frequency Cepstral Coefficients

Este método se basa en aplicar la Discrete Cosine Transform (Transformada Coseno Discreta) (DCT) a los valores de potencia, en escala logarítmica, del espectrograma de la señal con las frecuencias mapeadas según la escala de Mel. En resumen, sería aplicar la DCT al resultado del método *Mel Spectrogram*, mencionado anteriormente. Los Mel-Frequency Cepstral Coefficients (Coeficientes Ceptrales en las Frecuencias de Mel) (MFCC) son el resultado de las operaciones antes mencionadas.

Este método es utilizado para la extracción de características (features) en aplicaciones de reconocimiento del habla humana, como en reconocimiento de características musicales. En general presenta una buena robustez al ruido aditivo y logra comprimir información en un nivel considerable. Es por ello que este método se evalúa a los fines de realizar un pre-procesamiento más específico, buscando utilizar modelos de clasificador más sencillos (como los de ML tradicional). Un ejemplo del resultado de este método puede observarse en la figura 5.7e. Notar que la información se encuentra concentrada en los primeros coeficientes (parte baja de la imagen), por lo que podría trabajarse eventualmente con imágenes de menor tamaño.

El identificador de este método es *mfcc* y los parámetros de configuración correspondientes se muestran en el siguiente ejemplo:

```
{
"preprocessing_type": "mfcc",
"window": "hamming",
"wlen": 256,
"dynamic_range_dB": [70],
"mfcc_output": "mfcc"
}
```

El parámetro *mfcc_output* indica qué tipo de coeficientes se requiere. Si el valor es *mfcc* devuelve los coeficientes estándares, mientras que si el valor es *delta* o *delta2* devuelve la diferencias de primer y segundo orden correspondientemente, entre los coeficientes MFCC calculados, fragmento a fragmento de la señal.

5.5.6. 2D Cosine Transformed Spectrogram

Las DCT es una transformación basada en secuencias finitas obtenidas a través de suma de funciones coseno a diferentes frecuencias. Es una transformación relacionada con la DFT, pero que en este caso sólo utiliza números reales, realizándose una extensión periódica y simétrica de la señal. Este tipo de transformada es usada ampliamente en procesamiento de señales y en compresión de datos, por ejemplo, en imágenes JPEG, codecs de video como H.264 (MPEG-4) o audio como MP3.

Existen distintas variantes de esta transformada, dependiendo de cómo se haga la extensión de la señal. La más comúnmente usada es la DCT-II (léase "tipo dos"):

$$\boldsymbol{X}_k = \sum_{n=0}^{N-1} \boldsymbol{x}_n \cos\left[\frac{\pi}{N}\left(n+\frac{1}{2}\right)k\right] \quad ; k = 0, ..., N-1 \tag{5.33}$$

En este desarrollo se aplica una transformación multi-dimensional (2D), usando la DCT (específicamente la transformada del tipo II), al espectrograma $\boldsymbol{S}$ de la muestra. Esto equivale a aplicar la DCT por filas y luego por columnas, o viceversa, a la matriz que representa la imagen del espectrograma de la señal de un frame, calculado previamente. Entonces:

$$\boldsymbol{D}_{k_f,k_t} = \sum_{n_f=0}^{N_{pxf}-1} \sum_{n_t=0}^{N_{pxt}-1} \boldsymbol{S}_{n_f,n_t} \cos\left[\frac{\pi}{N_{pxf}}\left(n_f+\frac{1}{2}\right)k_f\right] \cos\left[\frac{\pi}{N_{pxt}}\left(n_t+\frac{1}{2}\right)k_t\right] \tag{5.34}$$

En la figura 5.7f se muestra un ejemplo del resultado de este método aplicado a la señal de un frame. El identificador de este método es *dct* y los parámetros de configuración correspondientes se muestran en el siguiente ejemplo:

```
{
"preprocessing_type": "dct",
"window": "hamming",
"wlen": 256,
"dynamic_range_dB": [100],
"dct_scale": "lin"
}
```

El parámetro *dct_scale* especifica cómo se hará el ajuste de escala de la imagen resultante de la DCT 2D. Si el valor es *lin* se realiza un proceso de estandarizado (se divide por la desviación estándar de los valores), se limita los valores a $|3\sigma|$, y se ajusta la escala para compatibilizar con el tipo de dato de cada píxel. Si el valor es *dB* se convierten los valores a deciBeles y luego se aplica el mismo ajuste de escala.

5.5.7. Escala de píxeles

Cada método de conversión de la señal Doppler a imágenes utiliza un método por defecto de conversión de los valores resultantes de su proceso a valor de intensidad de cada píxel de la imagen resultante. Esta conversión se implementó a través de una función denominada **spec_scaling**, definida como:

```
def spec_scaling(s, units='dB', normalize=True, dynamic_range=[], dtype=None,
        standarize=False, std_clip=None)
```

donde s es el producto resultante de la conversión realizada por cada método.

Esta función implementa un proceso de conversión de varias etapas:

Conversión de unidades Se realiza una conversión de unidades en función del parámetro *units*:

- `'dB'`: $20 \log_{10} |s|$
- `'power_dB'`: $10 \log_{10} |s|$
- `'abs'`: $|s|$
- `'lin'`: sin modificaciones

Estandarización Si el parámetro *standarize* es *True* se estandarizan los valores substrayendo su media y dividiendo por su desviación estándar:

$$s' = \frac{s - \bar{s}}{\sigma_s} \tag{5.35}$$

Adicionalmente, mediante el parámetro *std_clip*, se puede truncar los valores en función de su desviación estándar. Entonces, si a *std_clip* se le asigna el valor 2, se limitará los valores de s' al rango $[-2\sigma_s, +2\sigma_s]$.

Normalización Si el parámetro*normalize* es *True* se realiza una normalización de los datos al valor máximo de s. Dependiendo si la escala es lineal o en decibeles, tenemos:

- lineal: $s' = s / \text{máx}(s)$
- decibeles: $s' = s - \text{máx}(s)$

Rango dinámico Se selecciona una porción del rango dinámico que se mapeará al rango de intensidades de los píxeles de la imagen resultante. Existen tres alternativas de parametrización de *dynamic_range*:

- `'[ ]'`: Se mapea todo el rango dinámico de s.
- `'[`DR_{max}`]'`: conserva un rango dinámico de DR_{max} a partir del valor máximo de s. Entonces, si la escala el lineal el valor mínimo de s' será $\text{máx}(s)/\mathrm{DR}_{max}$; mientras que si la escala está en decibeles, el valor mínimo de s' será $\text{máx}(s) - \mathrm{DR}_{max}$.
- `'[`DR_{min}`, `DR_{max}`]'`: Se especifican arbitrariamente los valores mínimo y máximo de s que quieren mapearse.

Los valores que excedan los rangos que se mapean en la imagen resultante, se truncan al máximo o mínimo según corresponda.

Tipo de datos Finalmente, si se especifica un tipo de dato para la imagen resultante mediante el parámetro *dtype*, se realiza la conversión correspondiente haciendo:

$$s' = \frac{s - s_{min}}{s_{max} - s_{min}} \cdot (s'_{max} - s'_{min}) + s'_{min} \tag{5.36}$$

donde s_{min} y s_{max} son los valores mínimo y máximo de s resultante de las etapas anteriores; y donde s'_{min} y s'_{max} son los valores mínimo y máximo permitidos por el tipo de dato *dtype* (en el caso de *UINT8*, estos valores son 0 y 255).

5.6. Metadata

Junto a las particiones del Dataset, se construye una base de datos con la información adicional de cada muestra. Esta información permite realizar la trazabilidad completa de cada muestra a través de la *Dataset Chain*. Esta información se denomina **metadata** y es almacenada como una lista (cada ítem se corresponde con una muestra) de diccionarios de python. Un ejemplo de un elemento de esta lista, correspondiente a una muestra (frame), se muestra a continuación:

```
{'fs': 5681.818181818182,
'sample_id': 6649,
'signal_id': 119,
'segment_id': 0,
'segment_idx': array([     0, 127999]),
'segment_idx_t': array([ 0.      , 22.527824]),
'segment_len': 127999,
```

```
'segment_len_t': 22.527824,
'frame_id': 20,
'frame_idx': array([56840, 79568]),
'frame_idx_t': array([10.00384 , 14.003968]),
'frame_len': 22728,
'frame_len_t': 4.000128,
'filename': '1_9.mat',
'folder': '/media/data/learn/Maestría/Tesis/data/SPL/person/one/0/normal',
'path': '/media/data/learn/Maestría/Tesis/data/SPL/person/one/0/normal/1_9.mat',
'class': 'person',
'subclass': 'one',
'angle': '0',
'speed': 'normal',
'direction': '',
'hands': '',
'feet': '',
'equipment': '',
'misc': '',
'class_map_1': 'person',
'class_map_2': 'one',
'class_map_3': 'one',
'class_map_4': 'one',
'dataset_segment_id': 228,
'augmentation': '',
'augmentation_params': ''}
```


En esta estructura se puede recuperar información del frame sobre: frecuencia de muestreo, identificador de la señal de donde se obtuvo, identificador de su segmento, ubicación dentro del segmento, nombre del archivo original de la señal, duración, propiedades del blanco (features), clase a la que pertenece según mapeo, si fue generado en la etapa de data-augmentation, entre otras cosas.

Adicionalmente, se almacena información de cada mapeo generado, denominándose a esta estructura **maps**. Nuevamente, es una estructura tipo lista, con cada mapeo como elemento de esa lista. Cada mapeo tiene un diccionario con las siguientes etiquetas:

```
'y', 'y_map', 'train_idx', 'test_idx', 'class_prop'
```

Donde y es una lista de los índices de clase a la que pertenece cada muestra (enteros comenzando en 0), *y_map* almacena los nombre de cada clase asignado a cada índice de clase, *train_idx* es la lista de los índices de las muestras a usar para entrenamiento (partición Train), *test_idx* es la lista de los índices de las muestras a usar para la evaluación del desempeño (partición Test), *class_prop* es el nombre del conjunto de features usadas en ese mapeo.

5.7. Datasets para entrenamiento y evaluación

Durante el desarrollo del trabajo se han evaluado distintos tipos de configuraciones para la Dataset Chain, resultando en distintos Datasets para realizar el entrenamiento y la evaluación de los modelos. Algunos de los parámetros son comunes a todos los datasets y sus valores se han justificado anteriormente; entre estos valores podemos mencionar como ejemplos a: duración de cada frame (*frame_len*), tamaño de la muestra resultante (*output_dim*) o el tipo de dato usado para representar la intensidad de cada píxel (*dtype*). En la tabla 5.1 se muestra el detalle de los parámetros de configuración usados para algunos de los dataset evaluados.

El criterio general elegido para la conformación de los diferentes datasets ha sido tener una variedad de tipos de muestras, centrada en los distintos tipos de métodos de pre-procesamiento, como así también evaluar el impacto de no realizar data-augmentation (por ejemplo dataset *coronita*) y de tener un solapamiento menor de frames (por ejemplo dataset *corrida*). Se elige como dataset por defecto al denominado ***corona***, ya que es uno de los que presentó mejores resultados y menor demanda de capacidad de cómputo en el pre-procesamiento. Entonces, los resultados que se muestren en el resto del documento se corresponderán con este dataset, salvo que se indique lo contrario.

Para cada dataset los parámetros se han ajustado de manera manual utilizando herramientas (desarrolladas para este propósito) que permiten modificar los parámetros de manera interactiva y visual el resultado que produce la cadena. Vale mencionar que no se ha realizado optimización de estos parámetros de manera automática (optimización de hiper-parámetros).

		Datasets						
	Parameter name	**corona**	**coronita**	**escutoide**	**corrida**	**plaza**	**morlaco**	**morsa**
Dataset Params	frame_len	4.0	4.0	4.0	4.0	4.0	4.0	4.0
	frame_overlap	3.5	3.5	3.5	3.75	3.5	3.5	3.5
	window	hamming	hamming	hamming	hamming	hamming	hamming	hamming
	wlen	1024	1024	1024	1024	1024	1024	1024
	segm_threshold	25.0	25.0	25.0	20.0	25.0	25.0	25.0
	t_res	0.05	0.05	0.05	0.05	0.05	0.05	0.05
	segm_wlen	2.0	2.0	2.0	2.0	2.0	2.0	2.0
	max_gap_time	0.5	0.5	0.5	0.75	0.5	0.5	0.5
	min_segm_len	3.0	3.0	3.0	2.5	3.0	3.0	3.0
	add_noise	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
	segm_threshold_noise	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	t_res_noise	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	segm_wlen_noise	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	max_gap_time_noise	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	min_segm_len_noise	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	generate_noise_frames	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
	fs_noise	5681.8181	5681.8181	5681.8181	5681.8181	5681.8181	5681.8181	5681.8181
	noise_types	'uniform' 'gaussian'	'uniform' 'gaussian'	'uniform' 'gaussian'	'uniform' 'gaussian'	'uniform' 'gaussian'	'uniform' 'gaussian'	'uniform' 'gaussian'
	noise_num_signals_to_gen	20	20	20	20	20	20	20
	power_noise	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	bias_noise	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	t_len_noise	30.0	30.0	30.0	30.0	30.0	30.0	30.0
	train_size	0.7	0.7	0.7	0.7	0.7	0.7	0.7
	seed	0	0	0	0	0	0	0
Data Augmentation Params	frame_len	4.0	N/A	4.0	4.0	4.0	4.0	4.0
	frame_overlap	1.5	N/A	3.5	3.5	1.5	1.5	1.5
	augmentation_type	'time_dilation' 'time_contraction' 'noise_addition'	N/A	'time_dilation' 'time_contraction' 'noise_addition'	'time_dilation' 'time_contraction' 'noise_addition'	'time_dilation' 'time_contraction' 'noise_addition'	'time_dilation' 'time_contraction' 'noise_addition'	'time_dilation' 'time_contraction' 'noise_addition'
	mult_factor_of_max	1.2	N/A	1.2	1.2	1.2	1.2	1.2
	ratio_time_contraction	0.4	N/A	0.4	0.4	0.4	0.4	0.4
	ratio_time_dilation	0.4	N/A	0.4	0.4	0.4	0.4	0.4
	ratio_noise_addition	0.2	N/A	0.2	0.2	0.2	0.2	0.2
	max_contraction	0.5	N/A	0.5	0.5	0.5	0.5	0.5
	min_contraction	0.9	N/A	0.9	0.9	0.9	0.9	0.9
	max_dilation	2.0	N/A	2.0	2.0	2.0	2.0	2.0
	min_dilation	1.1	N/A	1.1	1.1	1.1	1.1	1.1
	SNR_max	50.0	N/A	40.0	50.0	50.0	50.0	50.0
	SNR_min	20.0	N/A	20.0	20.0	20.0	20.0	20.0
Preprocessing Params	preprocessing_type	fixdim	fixdim	fixdim	fixdim	multibanks	scalogram	dct
	output_dim	[128, 128]	[128, 128]	[128, 128]	[128, 128]	[128, 128]	[128, 128]	[128, 128]
	dtype	uint8	uint8	uint8	uint8	uint8	uint8	uint8
	decimation	4	4	4	4	4	4	4
	window	hamming	hamming	hamming	hamming	hamming	N/A	hamming
	wlen	256	256	256	256	320	N/A	256
	dynamic_range_dB	[100]	[100]	[70]	[70]	[100]	[60]	[100]
	base	N/A	N/A	N/A	N/A	6	N/A	N/A
	wavelet	N/A	N/A	N/A	N/A	N/A	cmor1-1.5	N/A
	level	N/A	N/A	N/A	N/A	N/A	200	N/A
	mfcc_output	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	dct_scale	N/A	N/A	N/A	N/A	N/A	N/A	lin

Tabla 5.1: Tabla de parámetros de configuración de Datasets. Se muestran los parámetros de configuración de los principales Datasets evaluados. Las celdas resaltadas muestran diferencias en los valores respecto al dataset *corona*.

Parte II

Clasificador

El **Clasificador** es el bloque encargado de tomar una muestra, ya con el formato adecuado, y realizar una inferencia, o clasificación, para esa muestra. La inferencia consiste en entregar un valor de probabilidad de que esa muestra pertenezca a una clase predefinida (ver 4.4 y 5.2), obteniéndose esta probabilidad para cada una de las clases predefinidas (mapeo). Una vez seleccionado el modelo entrenado más adecuado para el Clasificador, este bloque se despliega en la plataforma radar, formando parte de las cadenas de procesamiento que el mismo tenga, para la generación de productos para los usuarios del radar.

En este punto del desarrollo ya tenemos conformados los datasets que contienen las muestras que servirán para el entrenamiento y evaluación de los modelos que se propongan para la etapa de clasificación. Existen muchos métodos de ML (particularmente los de aprendizaje supervisado) que podrían aplicarse y evaluarse, que podrían llamarse métodos convencionales, tales como: Support-Vector Machines (Máquinas de Vector de Soporte) (SVM), k-Nearest Neighbors (k Vecinos más Cercanos) (k-NN), Linear Regression (Regresión Lineal) (LR), Decision Trees (Árboles de Decisión) (DT) o Neural Network (Red Neuronal) (NN) convencionales; pero uno de los objetivos para este desarrollo es utilizar métodos de DL, utilizándose en particular modelos del tipo CNN, motivado por el gran desempeño que han mostrado para clasificación de imágenes.

En esta parte del documento se hará una introducción a las CNN, la presentación de los modelos más relevantes para esta aplicación, se detallará el proceso de entrenamiento y evaluación (cadena de entrenamiento y evaluación, presentada en 3.3) y finalmente se mostrarán los resultados obtenidos para las distintas combinaciones de datasets, modelos y parámetros de entrenamiento.

Capítulo 6

Convolutional Neural Networks (CNNs)

Las **Redes Neuronales Convolucionales (Convolutional Neural Networks)**, o CNNs, son un tipo especializado de redes neuronales para el procesamiento de datos en una topología tipo grilla o matriz [1]. Este tipo de dato puede interpretase como una grilla 1-D, como por ejemplo una serie temporal; una grilla 2-D, como podría ser una matriz o una imagen bi-dimensional; o bien extender este concepto a más dimensiones, lo que se denomina *tensor* de manera genérica. Una imagen color presenta el ejemplo más común de tensor, ya que cada dimensión del color (Rojo, Vede, Azul), es una imagen bi-dimensional en sí, por lo que una imagen de estas características puede expresarse como un tensor 3-D cuyas dimensiones son $N_r \times N_c \times 3$, donde N_r y N_c son la cantidad de píxeles por fila y columna correspondientemente. Otro ejemplo de tensor podría ser una secuencia de imágenes, o video, en donde tendremos un tensor 4-D cuyas dimensiones serían $N_r \times N_c \times 3 \times N_t$, siendo N_t la cantidad de imágenes de la secuencia.

Este tipo de redes, fue presentada inicialmente como un modelo innovador para la clasificación de dígitos manuscritos por Le Cun en el año 1989 [67], y ha demostrado un gran desempeño en la clasificación de este tipo datos, especialmente de imágenes. Durante la década de 2010 se han dado importantes avances en el campo de DL, y en particular en la clasificación de imágenes usando CNNs. A partir de la conformación del dataset *ImageNet* [68], que contiene más de 14 millones de imágenes etiquetadas, con alrededor de 20.000 categorías, se realizó anualmente un desafío [69] para evaluar el desempeño de modelos de clasificación empleando este dataset. Ya el modelo *LeNet-5* [22] y algunos otros habían mostrado muy buenos resultados, pero no fue hasta la presentación del modelo *AlexNet* [23] (ver figura 6.1) para el desafío de *ImageNet*, que se logró mostrar resultados asombrosos para la clasificación de este tipo de imágenes. Se

necesitó de un dataset con una enorme cantidad de datos para lograr un buen desempeño en redes de este tipo, pero también de una capacidad de cómputo elevada, que en este caso se consiguió utilizando General Purpose Graphical Processing Unit (Unidad de Procesamiento Gráfico de Propósito General) (GPGPU). Estos importantes hitos dieron paso a una sucesión de desarrollos en el área que fueron mejorando los desempeños conseguidos, con modelos exitosos como: *VGG-16* [70], *ResNet* [71], *Inception* [72], *MobileNets* [73].

Estos excelentes resultados en la clasificación de imágenes ha motivado el uso de este tipo de modelos para realizar la clasificación de los blancos de interés, y como se dijo en la introducción de este documento, este trabajo se centra en la propuesta, entrenamiento y evaluación de modelos que utilizan CNNs para la clasificación de los ecos de blancos terrestres. En este capítulo se hará una introducción a la arquitectura CNN y a las partes que la componen, para luego presentar algunos de los modelos propuestos para la implementación del clasificador.

6.1. Arquitectura Genérica

Una red neuronal, o Artificial Neural Network (Red Neuronal Artificial) (ANN) [74], es uno de los tantos modelos de ML utilizados para implementar un clasificador; ver 6.6 para más detalle. En general, bajo un adecuado ajuste de sus hiper-parámetros y dataset, presentan un buen desempeño; sin embargo, los modelos tipo CNN han demostrado mejor desempeño y generalización que estas últimas para los mismos datasets (para más detalles ver las redes mencionadas en la introducción del capítulo).

En general, una CNN tiene una sola entrada (donde se colocará la imagen a clasificar) que será representada por un tensor, que para imágenes color tiene tres dimensiones $N_h \times N_w, \times N_{ch}$ (alto, ancho y número de canales), pero que para nuestra aplicación será de $N_h \times N_w \times 1$. La salida de la CNN será un tensor de dimensiones $N_{classes}$, o sea, un vector con las probabilidades asociadas a cada clase. Tener en cuenta que la cantidad de entradas y salidas puede variar dependiendo la necesidad y complejidad del modelo y que, como se explicó en la introducción, las dimensiones de las entradas y salidas también pueden variar en función de lo mismo.

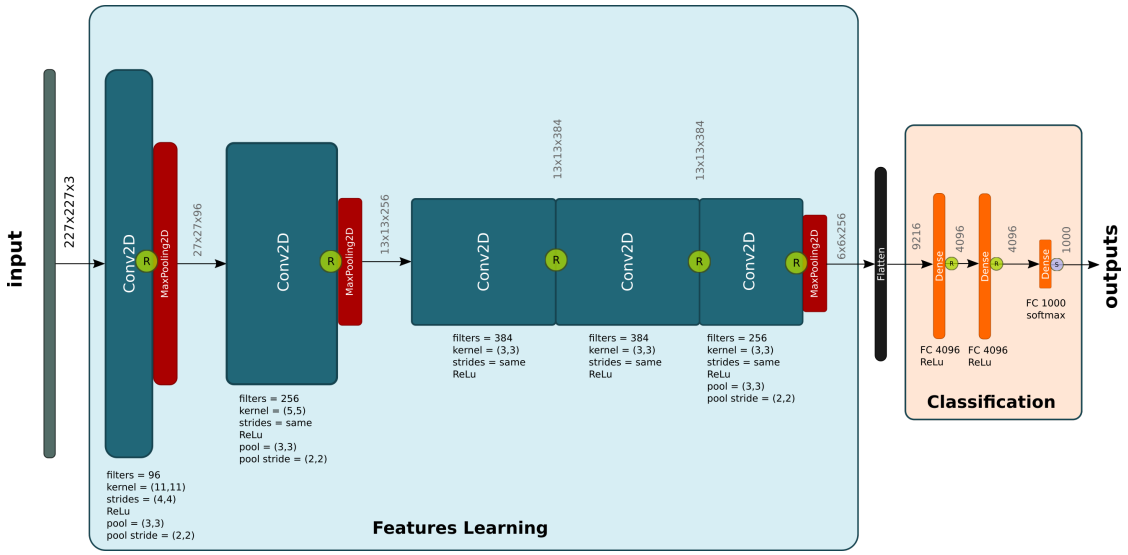


Figura 6.1: AlexNet - Arquitectura general del modelo.

La arquitectura genérica de una CNN puede presentarse dividida en dos grandes etapas: *aprendizaje de características (features learning stage)* y *clasificación propiamente dicha (classification stage)*. La etapa de **feature learning** está compuesta por una o más capas convolucionales en donde se aplican filtros que se desplazan a lo largo del tensor de entrada (de cada capa), donde los valores de cada uno de estos filtros se van ajustando durante el aprendizaje; el funcionamiento de estos filtros se explica en 6.3. El propósito de esta etapa es que los filtros se ajusten a los patrones más relevantes del conjunto de imágenes, generando los denominados *mapas de activación*, o dicho

de otro modo, se generan tensores que indican la presencia de dichos patrones en la entrada. Al final de esta etapa, se tendrá un tensor que albergará información relacionada con la distribución de patrones (patrones de patrones, en el caso de tener más de una capa convolucional) del tensor de entrada en función de cómo fue explorado. La etapa de **classification** está destinada a realizar una clasificación de la distribución de estos patrones (tensor de salida de la etapa de *features learning*). Se utilizan NN convencionales, que son redes densas de una o más capas y que en su capa de salida tendrá tantas neuronas como clases tenga el espacio de clasificación.

En la figura 6.1 se muestra el modelo *AlexNet*, como ejemplo de una CNN típica. En la imagen se resaltan las etapas de *Features Learning* y *Classification*. En la primera etapa se toma el tensor de entrada, que se corresponde a una imagen color de 227×227 píxeles (154.587 valores), y se aplican distintas capas convolucionales que van reduciendo el tamaño del tensor en la dimensiones de los píxeles, pero ampliando la cantidad de canales (que representan las activaciones de los distintos filtros). Al final de esta etapa se dispone de un tensor de dimensiones $6 \times 6 \times 256$ (9.216 valores), por lo que se realiza una compresión de información. Este tensor 3D se convierte a un tensor unidimensional (Flatten) para convertirse en la entrada de la etapa de *Classification*. Esta última etapa está conformada por una NN densa de 2 capas ocultas de 4096 unidades (neuronas) y una capa de salida de 1000 unidades que se corresponde con la cantidad de clases del dataset.

6.2. Capa de Entrada

Se denomina *Input Layer (capa de entrada)* al tensor que contiene los datos originales a clasificar. Si bien, recibe el nombre de "capa"no es una parte de la red propiamente dicha. La dimensión de la entrada, como se dijo anteriormente, es $N_h \times N_w, \times N_{ch}$ (número de píxeles en altura, ancho y cantidad de canales), pero que usualmente se expresa en los modelos con una dimensión extra N_{batch}, que representa la cantidad de muestras que contiene un Batch, ver 7.1. La dimensión de la entrada es un hiper-parámetro invariable para el modelo.

Si bien se mencionó que los datos que se ingresan por la *Input Layer* son los datos originales, en la mayoría de los casos es necesario realizar una *normalización* o *estandarización* de los mismos; buscando, dependiendo del método, que la distribución de los valores se encuentren en el rango de $(-1, 1)$ o $(0, 1)$, considerando valores mínimos/máximos o desviación estándar. Esta operación permite distribuir los valores en el rango de mayor variación de las funciones de activación.

6.3. Convolución

En terminos generales, la operación convolución entre dos funciones x y w se expresa como:

$$s(t) = (x * w)(t) = \int x(\tau)w(t-\tau)d\tau = \int x(t-\tau)w(\tau)d\tau \tag{6.1}$$

Esta función integra el producto de una función x con la versión espejada de w, o viceversa, que puede verse también como el área de la intersección de ambas funciones para cada valor de desplazamiento. Conceptualmente representa un indicador de similitud de ambas funciones (una de ellas espejada) para un desplazamiento dado. [1].

En redes convolucionales, el primer argumento (x en el caso de la ecuación 6.1) se denomina **input** (entrada) de la capa, mientras que el segundo argumento (w para la misma ecuación) se denomina **kernel** [1]. La salida usualmente se denomina **feature map** (mapa de característica).

$$s[n] = (x * w)[n] = \sum_m x[m]w[n-m] = \sum_m x[n-m]w[n] \tag{6.2}$$

Notar que la ecuación 6.1 define la convolución para el caso continuo, tanto para los dominios de las variables como para los desplazamientos, y para el caso unidimensional. Para el caso discreto, y desplazamientos acotados, la convolución se expresa como 6.2 que se acerca más a lo que se utilizará en aplicaciones de ML.

En el caso de aplicaciones de Machine Learning, la *entrada* a una capa convolucional es un arreglo multidimensional (*tensor*), y el *kernel* es también un tensor de parámetros que definen un filtro cuyos valores serán adaptados por el algoritmo de aprendizaje. Si en la aplicación la entrada es una imagen $\boldsymbol{I}$ bi-dimensional, con un determinado número de canales, se usará un *kernel* $\boldsymbol{K}$ con igual cantidad de dimensiones [1] que la imagen para realizar la operación de convolución. El Kernel se "desplazará"horizontal y verticalmente para recorrer toda la imagen. La convolución la podemos escribir como:

$$\boldsymbol{S}[i,j] = (\boldsymbol{K} * \boldsymbol{I})[i,j] = \sum_m \sum_n \boldsymbol{I}[i-m, j-n]\boldsymbol{K}[m,n] \tag{6.3}$$

Las librerías de ML generalmente implementan la convolución sin espejar la imagen o el Kernel, denominándose específicamente **cross-correlation**. Adicionalmente a esto se agrega un peso más denominado **bias** b, entonces:

[1] El termino igual "cantidad de dimensiones" se refiere a que los tensores tienen igual cantidad de componentes, pero no necesariamente estas componentes deben tener tamaños iguales. En las CNNs, la dimensión correspondiente a los canales (color) sí tiene igual tamaño en I y K.

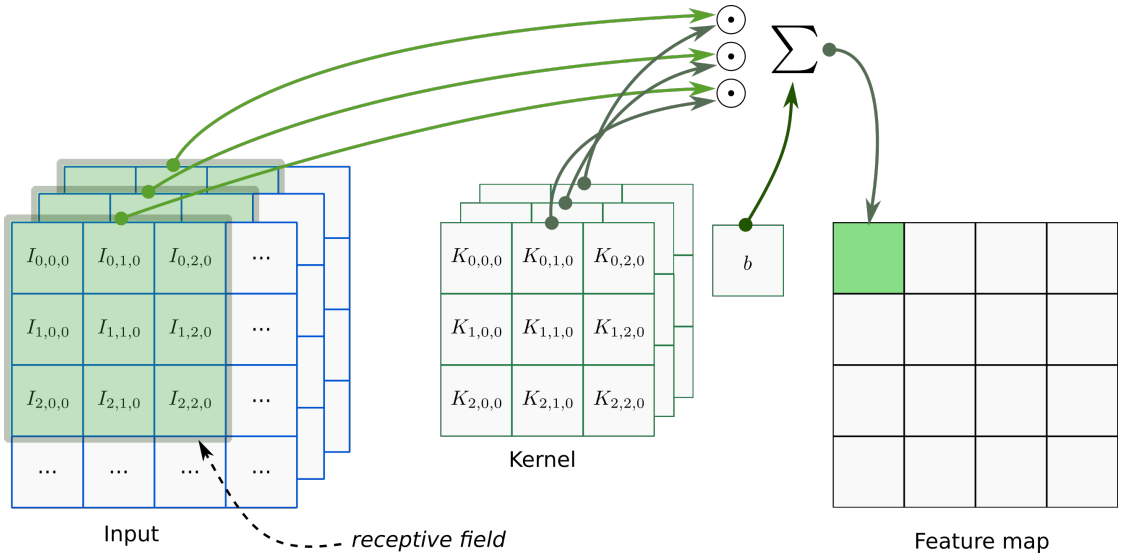


Figura 6.2: Convolución entre una imagen 2D con 3 canales y un Kernel de 3×3. Notar que el Kernel tiene la misma cantidad de canales que la imagen. Se realiza un producto interno entre elementos de una porción de la imagen (definida por el desplazamiento) y los elementos del Kernel obteniéndose el valor de convolución para ese desplazamiento.

$$\boldsymbol{S}[i,j] = (\boldsymbol{K} * \boldsymbol{I})[i,j] = \sum_m \sum_n \boldsymbol{I}[i+m, j+n]\boldsymbol{K}[m,n] + b \tag{6.4}$$

Básicamente, la operación de convolución en una imagen se realiza tomando una porción de la imagen con el mismo tamaño del Kernel, y aplicar el producto interno entre los valores de ambos tensores, para obtener un valor escalar que se corresponderá a ese Kernel y a ese desplazamiento en particular. En la figura 6.2 se muestra un ejemplo de esta operación. La porción de la imagen usada para realizar la operación de convolución se denomina **receptive field** (campo receptivo). El tamaño del Kernel es un parámetro de la capa convolucional. Es importante hacer notar que los valores del Kernel que se van adaptando con el entrenamiento son constantes a medida que se realizan los desplazamientos, por lo que la cantidad de pesos (valores del kernel) a entrenar es independiente del tamaño del tensor de entrada.

La operación de convolución entregará tantos valores escalares por cada desplazamiento del Kernel que se realice sobre el tensor de entrada, por lo que este conjunto de valores define el *feature map* correspondiente a ese Kernel. En la figura 6.3 se muestra un ejemplo del desplazamiento (en este caso horizontal) del receptive field lo que genera un valor nuevo en el feature map. El desplazamiento, tanto horizontal como vertical, es un parámetro de la capa convolucional y puede ser distinto de 1 inclusive. El paso de desplazamiento se define como **stride**, pudiendo ser diferente para el desplazamiento horizontal y vertical. El tamaño del tensor de entrada (alto y ancho), el tamaño del Kernel y el desplazamiento definen el tamaño del tensor de salida (feature map), según 6.5. En ocasiones es conveniente agregar filas y/o columnas con ceros (*zero padding*)

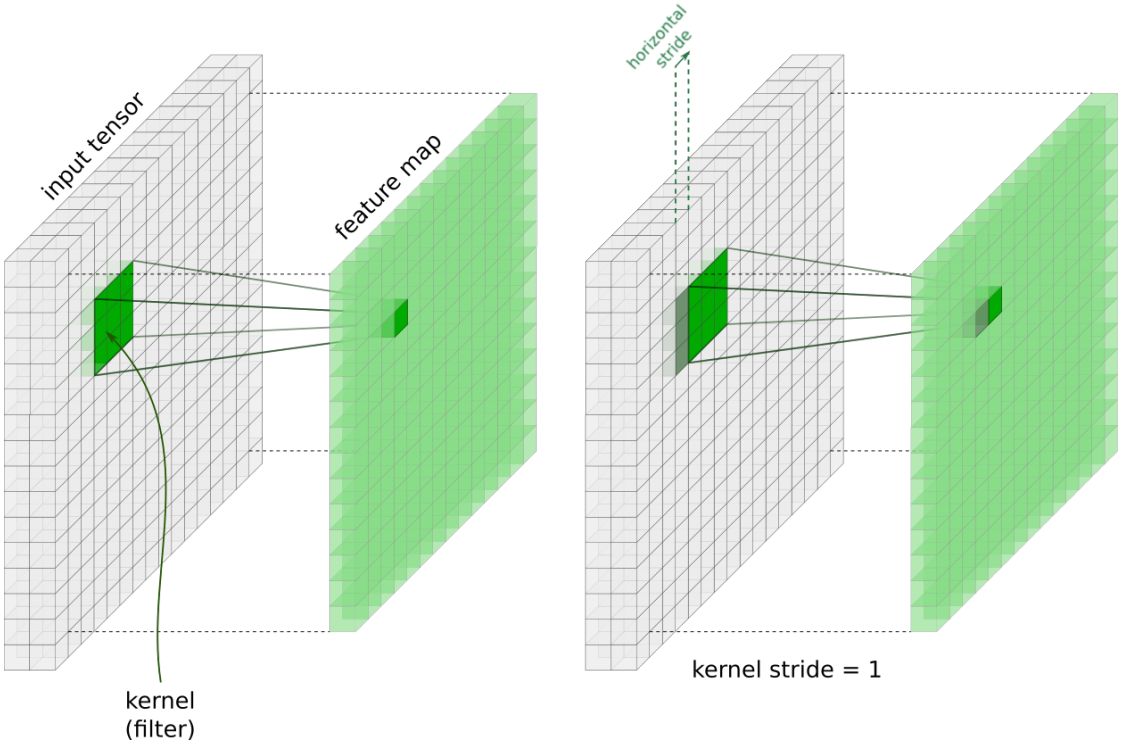


Figura 6.3: Convolución, desplazamiento del Kernel a lo largo del tensor de entrada. Por cada desplazamiento se tiene un valor escalar distinto en el mapa de característica.

para ajustar las dimensiones de salida.

$$N_o = \frac{N_i - F + 2P}{S} + 1 \tag{6.5}$$

siendo N_o la cantidad de elementos de salida, N_i la cantidad de elementos de la entrada para la misma dimensión, F el tamaño del Kernel, P la cantidad de elementos agregados por el padding en cada extremo y S el paso de desplazamiento (stride). Todos los valores se corresponden con la dimensión que se esté recorriendo. Como puede deducirse, el stride tiene un fuerte impacto en la compresión de información, y en segundo lugar el tamaño del Kernel.

Una capa convolucional puede tener más de un Kernel (Filtro), como se muestra en la figura 6.4, con la restricción de que todos los Kernel deben tener el mismo tamaño, con iguales parámetros de desplazamiento, para asegurar que los feature maps sean de igual tamaño. La cantidad de kernels de una capa convolucional se denomina **depth** (profundidad), y se corresponderá con la cantidad de canales de salida del tensor de salida de dicha capa. Al finalizar el entrenamiento, los valores de cada Kernel definen un filtro adaptado a una característica o patrón particular, y es por ello que la profundidad de una capa convolucional define la cantidad de características distintas que se podrán diferenciar dentro del tensor de entrada. La cantidad de pesos a entrenar en una capa convolucional será entonces:

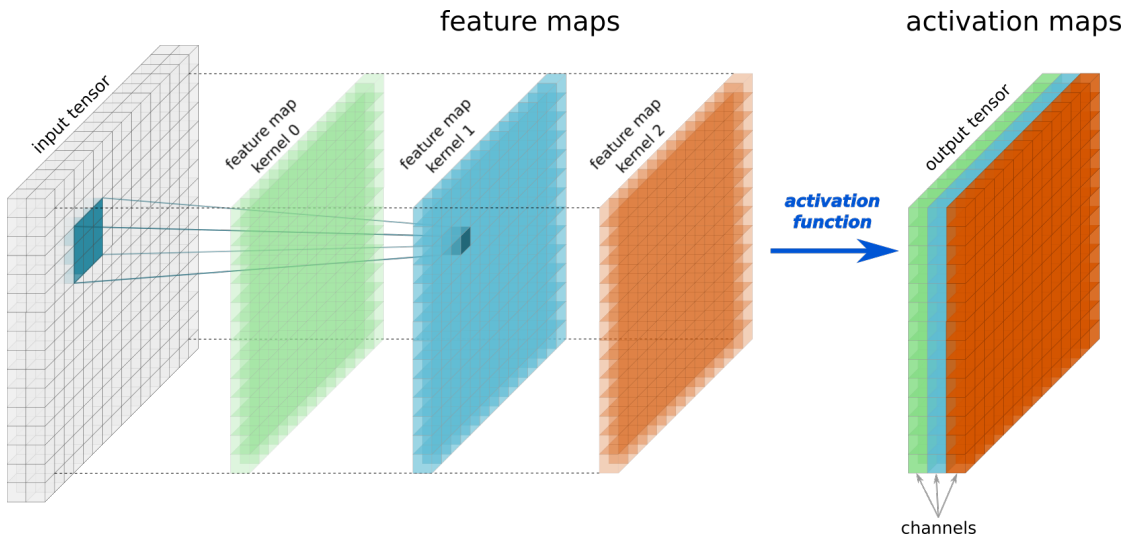


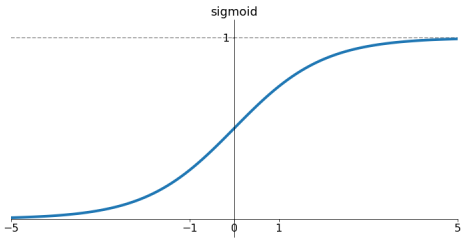
Figura 6.4: Mapas de activación. Cada una de las salidas de los Kernels (feature map) es afectada por la función de activación definida para esa capa convoluciones, generando por cada salida un *activation map*, que representará un canal en el tensor de salida de esa capa convolucional.

$$N_w = (F_h \cdot F_v + 1)D \tag{6.6}$$

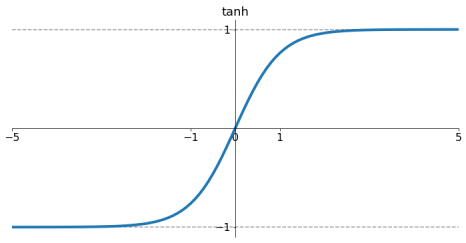
donde F_h y F_v es el tamaño de cada Kernel en la dimensión horizontal y vertical correspondientemente, y D es la profundidad de la capa (cantidad de Kernels). Se agrega por cada Kernel un valor de **bias** según 6.4, que permite agregar un peso adicional para ajustar un offset a la salida de cada Kernel.

6.4. Activación

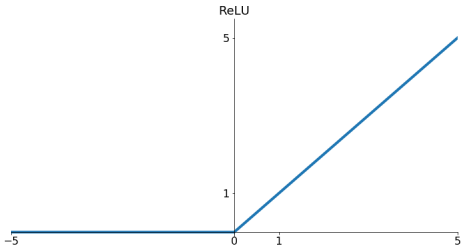
La operación realizada por el Kernel en una capa convolucional es equivalente a la realizada en las redes neuronales convencionales [74], pero en el caso convolucional sería sólo con conectividad local (receptive field). Podemos decir entonces que por cada Kernel de una capa convolucional hay tantas neuronas como campos receptivos se hayan definido sobre la entrada a esa capa (definidos por los desplazamientos del Kernel); y que cada una de estas neuronas comparten los mismos pesos. Entonces, al igual que en las redes neuronales convencionales, la salida de la combinación lineal entre los valores de entrada y los pesos de la capa se pasa como argumento a una función no-lineal que se denomina **activation function** (función de activación). Cada feature map (salidas de un Kernel) excitará la función de activación definida para esa capa, resultando en un **activation map** (mapa de activación), como se observa en la figure 6.4. El uso de funciones no-lineales está asociado a la capacidad de poder aproximar una función de una variable (usada para la separación entre clases) con varias variables (entradas), ver [75]. Cada neurona tendrá una salida dada por:



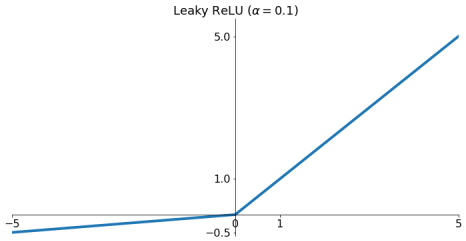
(a) sigmoid



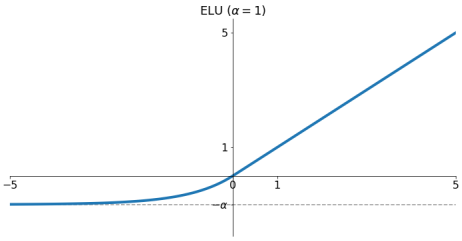
(b) tanh



(c) ReLU



(d) Leaky ReLU



(e) ELU

Figura 6.5: Funciones de activación comúnmente usadas.

$$y(\boldsymbol{x}) = g\Big(\sum_i w_i x_i + b\Big) \tag{6.7}$$

donde $g()$ es la función de activación, w y b los pesos de y bias asociados a la neurona, x los valores de entrada, e y el valor de salida de la neurona para una entrada dada, también denominado valor de **activación**.

Existe una variedad grande de funciones de activación [76, 77], donde la selección depende del tipo y ubicación de cada capa de la red neuronal y también de la aplicación. La figura 6.5 muestra las funciones de activación comúnmente usadas en aplicaciones de ML y que algunas de ellas fueron usadas en los modelos evaluados para esta aplicación. A continuación se presentan brevemente estas funciones:

Sigmoid Esta función está definida como $\sigma(x) = 1/(1+e^{-x})$ y se muestra en la figura 6.5a. En rasgos generales limita su salida a una rango entre 0 y 1, limitando los valores negativos grandes a 0 y los grandes valores positivos a 1. Esta función fue muy utilizada

en redes neuronales convencionales debido a la similitud de comportamiento con las neuronas reales; pero que en los modelos actuales es raramente usada. Las desventajas de esta función son:

- *Satura y mata el gradiente*: Cuando esta función entrega valores saturados, ya sea en 0 o 1, el gradiente en estas regiones se hace casi cero (derivada de la función en estas regiones). Al aplicar backpropagation durante el aprendizaje, el hecho que el gradiente sea casi cero insensibiliza la propagación del error hacia los pesos de esa neurona y de las de las capas anteriores, "matando" el gradiente.
- *Salidas no centradas en cero*: El hecho de que las salidas son siempre positivas, entre 0 y 1, hace que la distribución de los datos pasados a la capa siguiente no estén centrados en cero, lo que puede afectar a la dinámica del descenso por gradiente, generalmente del tipo zig-zag.

Tanh La tangente hiperbólica, mostrada en la figura 6.5b, limita la excursión de su salida entre -1 y $+1$, lo que es muy similar a la función *sigmoid*, pero ahora la salida sí está centrada en cero. Es por ello, que en general se prefiere usar esta función en lugar de la sigmoid. Notar que esta función puede escribirse en función de la sigmoid como: $\tanh(x) = 2\sigma(2x) - 1$.

ReLU La función Rectified Linear Unit (Unidad Lineal Rectificada) (ReLU) computa la función $g(x) = \max(0, x)$. Básicamente, limita sólo los valores negativos de la entrada a 0. Esta función es una de las más utilizadas en los modelos actuales, y más aún en los modelos de DL, presentando las siguientes ventajas y desventajas:

- (+) Se encontró que acelera la convergencia usando descenso por gradiente estocástico, en comparación con las funciones sigmoid y tanh, basado en su comportamiento lineal, no saturado.
- (+) Se implementa con menor costo computacional, ya que las funciones sigmoid y tanh utilizan funciones más costosas, como exponenciales e inversiones.
- (-) Las unidades que utilizan esta función de activación pueden ser frágiles durante el entrenamiento y pueden "morir". Un ejemplo de esto se da cuando un gradiente muy grande fluye a través de una unidad con ReLU, causando que sus pesos se actualicen de tal manera que esa neurona no se activará nuevamente independientemente de los datos de entrada. Si esto ocurre, el gradiente a través de esta unidad, y en adelante, será cero durante el resto del entrenamiento. Generalmente esto sucede cuando se usan *learning rates* (tasas de aprendizaje) muy grandes.

Leaky ReLU Estas funciones aparecen como una modificación de las ReLU para arreglar el problemas de las unidades que "mueren" durante el entrenamiento. Entonces, en lugar de hacer 0 los valores negativos a la entrada, los afecta por una pequeña pendiente definida por un factor α, como se muestra en la figura 6.5d. Esta función puede escribirse como $g(x) = \mathbf{1}(x < 0)(\alpha x) + \mathbf{1}(x \geq 0)(x)$, donde α suele ser una constante pequeña (0,01 por ejemplo).

Maxout Una alternativa que generaliza las funciones ReLU y Leaky ReLU es la función Maxout, que se computa como $g(x) = \text{máx}(w_1^T x + b_1, w_2^T x + b_2)$ (notar que la ReLU es un caso especial de esta función al hacer $w_1, b_1 = 0$). Esta función, entonces, tiene las ventajas de la ReLU y Leaky ReLU, pero presenta la desventaja de duplicar la cantidad de parámetros por cada unidad, incrementando la cantidad de pesos del modelo a entrenar.

ELU La función Exponential Linear Unit (Unidad Lineal Exponencial) (ELU) también soluciona el problema de la "muerte" de la ReLU tal como lo hace la Leaky ReLU, pero a diferencia de esta última, en lugar de agregar una zona lineal para los valores negativos, computa una exponencial, tal como se muestra en la figura 6.5e. Esta función se define como: $g(x) = \mathbf{1}(x < 0)(\alpha(e^x - 1)) + \mathbf{1}(x \geq 0)(x)$, donde el parámetro α suele elegirse entre 0,1 y 0,3. Para grandes valores negativos, esta función satura en el valor $-\alpha$, por lo que es una desventaja desde el punto de vista del desvanecimiento del gradiente. Otra desventaja es que es computacionalmente más costosa, debido a que debe computar la función exponencial. Sin embargo, como ventajas se pueden mencionar que presenta una transición más suave entre valores de entrada negativos y positivos lo que ayuda en el ajuste de los pesos de la red durante el entrenamiento, consiguiendo una mejor y más rápida minimización de la función de costo.

6.5. Pooling

La capa de *Pool* (o capa de *Pooling*), realiza un sub-muestreo sobre las dimensiones espaciales (alto y ancho en el caso de imágenes bi-dimensionales) del tensor de entrada a dicha capa, aplicando un filtro que se desplaza recorriendo el tensor, de la misma manera que lo hace la capa convolucional. Esta operación se denomina **Pooling** y está destinada a reducir el tamaño de las dimensiones que se recorran con el filtro. A diferencia de la capa convolucional, el filtro no actúa sobre todos los canales del tensor de entrada a la vez, sino que se aplica por cada canal de manera independiente, por lo que la capa de *Pooling* entrega un tensor con la misma cantidad de canales en su salida.

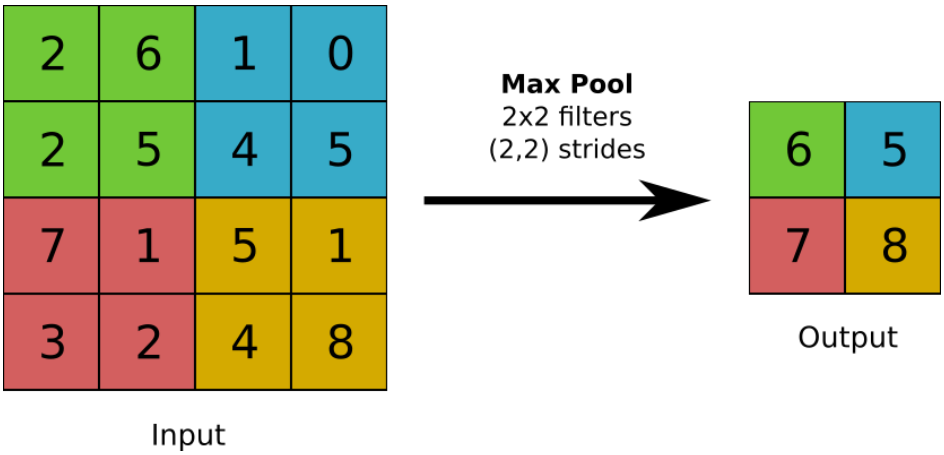


Figura 6.6: Max Pooling. Aquí se muestra un sub-muestreo de una matriz de 4×4 con un filtro de 2×2, con pasos de 2 muestras (horizontales y verticales), usando el método de elegir el máximo valor de las muestras ingresadas al filtro.

Es común, en las redes convolucionales, insertar una capa de Pooling entre sucesivas capas convolucionales para ir reduciendo el tamaño espacial, como se mencionó, lo que beneficia en la reducción de parámetros del modelo y en la reducción del *overfitting* (sobreajuste). La función más común para realizar el pooling es la función máx, que elije el mayor valor del campo receptivo del filtro; y en tal caso la operación se denomina **max pooling**. Alternativamente se pueden usar otras funciones como *average pooling* (promediado de los valores) y *L2-norm pooling* (norma L2 de los datos).

En la figura 6.6 se muestra un ejemplo de la operación max pooling usando un filtro de 2×2 y desplazamiento de 2 tanto para la dirección horizontal como vertical (stride igual a 2), por lo que los campos receptivos no se solapan; generando una decimación de 2 en cada dirección sobre el tensor de entrada, o bien de 4 en la cantidad de datos. Este tipo de configuración es la más usual en las redes convolucionales. Una alternativa usual es usar un tamaño de filtro de 3×3 pero con el stride igual a 2, lo que genera un solapamiento de los campos receptivos. Para calcular el tamaño del tensor de salida de una capa de pooling, tenemos:

$$N_o = \frac{N_i - F}{S} + 1 \tag{6.8}$$

siendo N_o la cantidad de elementos de salida, N_i la cantidad de elementos de la entrada para la misma dimensión, F el tamaño del filtro, S el paso de desplazamiento (stride). Recordar que una capa de pooling no altera la profundidad del tensor de entrada, por lo que la salida mantiene la misma cantidad de canales. Notar que, a diferencia de la ecuación 6.5, aquí no se considera el padding, ya que no es común hacerlo en este tipo de capas.

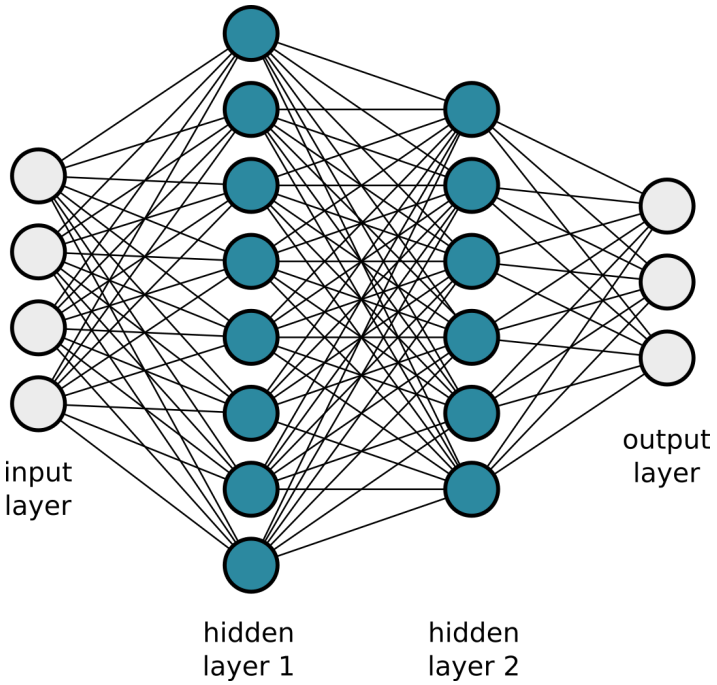


Figura 6.7: Red neuronal completamente conectada, también conocida como *Dense* o *Fully-Connected*. En este ejemplo se muestra una capa de entrada, dos capas ocultas y una capa de salida. Cada línea que vincula a las neuronas representa un peso a ajustar durante el entrenamiento.

6.6. Redes Densas

El tensor de salida de la última capa convolucional tendrá en cada dato un valor que representa la presencia de patrones complejos (o patrones de patrones) de las distintas características definidas por los filtros de las capas convolucionales anteriores, y que se corresponde con un campo receptivo acotado del tensor de entrada. Esto quiere decir que la distribución espacial de estas activaciones en la salida tiene correlación con la distribución espacial de la entrada. En esta instancia es necesario realizar la clasificación de estos patrones complejos a lo largo de todo el tensor, y es por ello que se prefiere el uso de redes completamente conectadas para la etapa de clasificación. Entonces se convierte el tensor de salida de la etapa de *feature learning* a un vector unidimensional que será la entrada de la red de clasificación.

Una red neuronal tipo **Dense** (Fully-connected) [1, 74], o *completamente conectada*, como la que se muestra en la figura 6.7, se compone de: una capa de entrada, que no es más que el vector con los datos de entrada; capas ocultas en donde la cantidad de capas y la cantidad de neuronas por capa son hiper-parámetros configurables; y una capa de salida que tiene la cantidad de salidas igual a la cantidad de clases sobre la que se quiere realizar la clasificación (ver 6.7). Cada neurona de una capa realiza la operación mostrada en la ecuación 6.7, pero ahora el campo receptivo es toda la entrada a la capa (salida de la capa anterior). Se puede entonces calcular todas las salidas de una

capa realizando una operación matricial, tal como:

$$\boldsymbol{h} = g(\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b}) = g(\boldsymbol{W'}^T\boldsymbol{x'}) \tag{6.9}$$

donde $\boldsymbol{W}$ son los pesos correspondientes a dicha capa, $\boldsymbol{x}$ la entrada y $\boldsymbol{b}$ los valores de bias. Notar que también se puede incluir al bias como pesos expandiendo la entrada con la constante 1, formando $\boldsymbol{x'}$.

Debido a que cada neurona toma toda la entrada a la capa, y que una capa suele tener una cantidad grande de neuronas (buscando un modelo con mayor capacidad de clasificación), la cantidad de pesos a ajustar en una red de este tipo es muy grande; donde por cada capa, la cantidad de pesos a ajustar es:

$$N_w = (N_i + 1) * N_L \tag{6.10}$$

donde N_w es la cantidad de pesos a ajustar en una capa determinada, N_i es la cantidad de datos de entrada a esa capa y N_L es la cantidad de neuronas de la capa. Notar que la suma en 1 a N_i es porque se considera el valor de bias de cada neurona.

Otro de los hiper-parámetros a definir en este tipo de red es la función de activación $g()$, que se define de manera independiente por capa de la red. Históricamente se utilizaba las funciones de activación *sigmoid* y *tanh*, pero han sido mayormente reemplazadas por la función *ReLU*.

6.7. Capa de Salida

En los modelos destinados a realizar una clasificación del tipo multi-clase, la salida del modelo tiene la misma dimensión que la cantidad de clases sobre las que se quieren clasificar las muestras de entrada. La salida será, entonces, un vector donde cada componente se corresponde con una clase en particular. En las redes convolucionales, como la última etapa (clasificación) está conformada por una red densa multi-capa, la capa de salida es una capa completamente conectada con la última capa oculta.

Para que los valores de salida del modelo "representen" un valor de probabilidad, la función de activación utilizada en cada una de las neuronas de la capa de salida debe tener su rango de salida limitado a $[0, 1]$, como por ejemplo la función *sigmoid*. Pero, en muchas de las aplicaciones se requiere que la suma de todas las componentes de salida (probabilidades de todas la clases) sea igual a 1. Esto se consigue utilizando la función **Softmax**, definida como:

$$\sigma(\boldsymbol{x})_j = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}} \quad j = 1, ..., K \tag{6.11}$$

donde K es la dimensión del vector de salida, o bien, la cantidad de clases.

6.8. Capa de Regularización

Cuando se detalló la *Dataset Chain* (3.2) se explicó la conveniencia de incrementar la cantidad de datos de entrenamiento para mejorar el desempeño del modelo para datos nuevos (datos no usados para el entrenamiento), lo que se denomina capacidad de generalización del modelo. Esto es un problema central en los algoritmos de ML, en donde se han encontrado diversas estrategias para mejorar la generalización de los modelos, conocidas como estrategias de **regularización**. [1, 77–79].

A diferencia de la estrategia *Data-Augmentation* aplicada en la *Dataset Chain*, se pueden aplicar estrategias de regularización dentro del mismo modelo. Estas estrategias pueden orientarse a poner restricciones al modelo, ya sea en su arquitectura (modelos más simples tienden a generalizar mejor) o en los valores de sus parámetros; como también agregar términos adicionales a la función objetivo o función de costo que se quiere minimizar.

En esta sección se presentarán las estrategias usadas para realizar una regularización del modelo, pero que están vinculadas estrechamente a la arquitectura del mismo, mientras que las vinculadas a la función de costo se presentarán en la sección 7.4. En los modelos propuestos y evaluados en este trabajo se usaron dos tipos de estrategias de regularización: *Dropout* y *Batch Normalization*. Ambas estrategias se representan como capas del modelo, pues aplican un método de regularización en una etapa en particular del modelo, pudiéndose usar más de una capa de regularización si fuera deseable.

6.8.1. Dropout

Esta estrategia de regularización [80] que se basa en la modificación de la capacidad del modelo apagando unidades (neuronas) de las capas en donde se aplique. El apagado de las unidades se hace aleatoriamente durante el entrenamiento del modelo, donde la probabilidad de que una unidad en particular se apague es el hiper-parámetro de la capa de *Dropout*. Una de las principales ventajas de esta estrategia es que es muy barata desde el punto de vista de costo computacional.

En la figura 6.8 se muestra un ejemplo de Dropout aplicado a la primera capa oculta, donde una determinada proporción de neuronas se encuentran apagadas (50 %

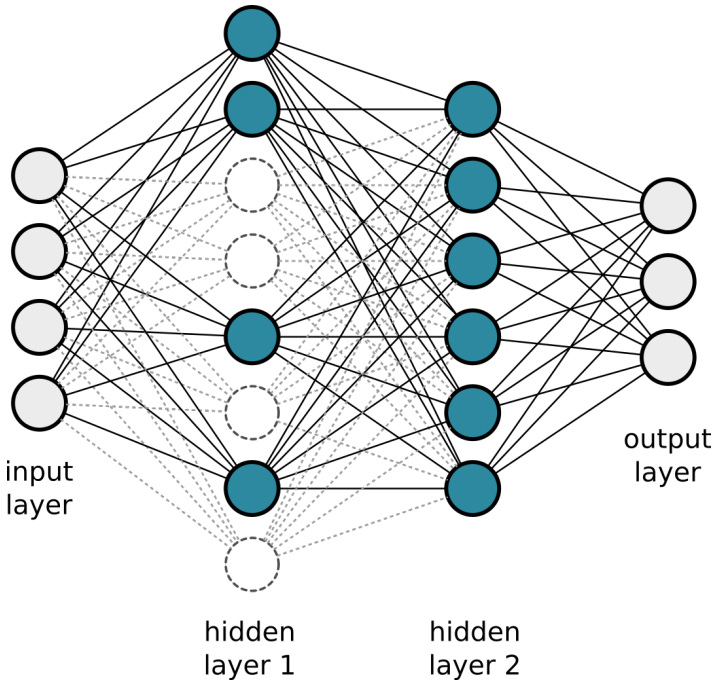


Figura 6.8: Dropout. Aplicación de un dropout del 50 % a la primer capa oculta de una red densa.

en este caso). Esta configuración de la red se utilizará para ajustar los pesos para una cierta cantidad de muestras de entrenamiento, definidas como *mini-batch*. Para el próximo mini-batch de muestras, se restablecen las neuronas apagadas y nuevamente se aplica el apagado aleatorio para las neuronas de esa capa. En la práctica, el Dropout se aplica a las capas internas del modelo, sean convolucionales o densas; también se puede aplicar a la capa de entrada, pero no a la capa de salida.

Al aplicar Dropout a un modelo, su arquitectura podría verse como la de múltiples redes (de menor capacidad) que se están entrenando en paralelo con los mismos datos de entrenamiento. Esto es similar a una estrategia de regularización denominada *Bagging*, en donde se entrenan diferentes modelos en paralelo y luego se hace un promediado de los resultados. El principio de funcionamiento se basa en que todas las redes entrenadas en paralelo sufrirán de overfitting en diferentes "direcciones", y que al promediar los resultados se espera minimizar el error de generalización del modelo completo. Otra forma de ver el Dropout es como un robustecimiento del modelo ante la pérdida parcial de información. [1, 77].

6.8.2. Batch Normalization

Es un método utilizado para acelerar el aprendizaje y mejorar la estabilidad de las redes neuronales a través de la normalización de los datos de entrada a una capa, centrándolos usando su media y cambiando su escala en función de su varianza. [81].

Este método reduce la dependencia entre las capas durante el aprendizaje, permitiendo el uso de tasas de aprendizaje mayores, lo que aceleraría el entrenamiento. También reduce el overfitting ya que tiene un efecto de regularización, e incluso puede agregarse ruido luego de la normalización propiamente dicha para acentuar este efecto. En general se usa en conjunción con Dropout, permitiendo reducir el nivel de Dropout aplicado a la capa. [82].

El proceso de normalización se realiza a nivel mini-batch, al igual que el Dropout, por lo que la media y varianza usadas en el proceso de normalización se calculan usando los datos de entrada a la capa correspondientes a un mini-batch. La normalización se suele aplicar antes de la capa de activación. Este proceso de normalización se muestra en las siguientes ecuaciones:

$$\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{6.12}$$

$$\sigma^2_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \tag{6.13}$$

$$\hat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \tag{6.14}$$

donde x_i son los valores de entrada a la capa correspondientes al mini-batch $\mathcal{B}$, $\mu_{\mathcal{B}}$ es la media y $\sigma^2_{\mathcal{B}}$ la varianza de dichos datos (ϵ es una constante agregada para la estabilidad numérica), y $\hat{x}_i$ los datos normalizados. Sin embargo, esta normalización puede afectar a la distribución de los pesos de la próxima capa; es por ello que se realiza una des-normalización utilizando un factor de escala γ y de bias β (equivalentes a una desviación estándar y media) que serán ajustados durante el proceso de aprendizaje buscando la minimización de la función de costo. Entonces, la salida de la capa de *Batch Normalization* (BN) se expresa como:

$$y_i = \gamma\hat{x}_i + \beta \triangleq BN_{\gamma,\beta}(x_i) \tag{6.15}$$

6.9. Modelos propuestos

Durante el desarrollo de este trabajo se han evaluado diversos modelos, muchos de ellos inspirados en trabajos afines, ya sea de clasificación de imágenes convencionales, o bien en clasificación de espectrogramas de señales de audio, [28, 29]. En esta sección se presentan algunos de los modelos evaluados, seleccionados en función de sus características diferenciadoras y del desempeño obtenido para los datasets. Simplemente se

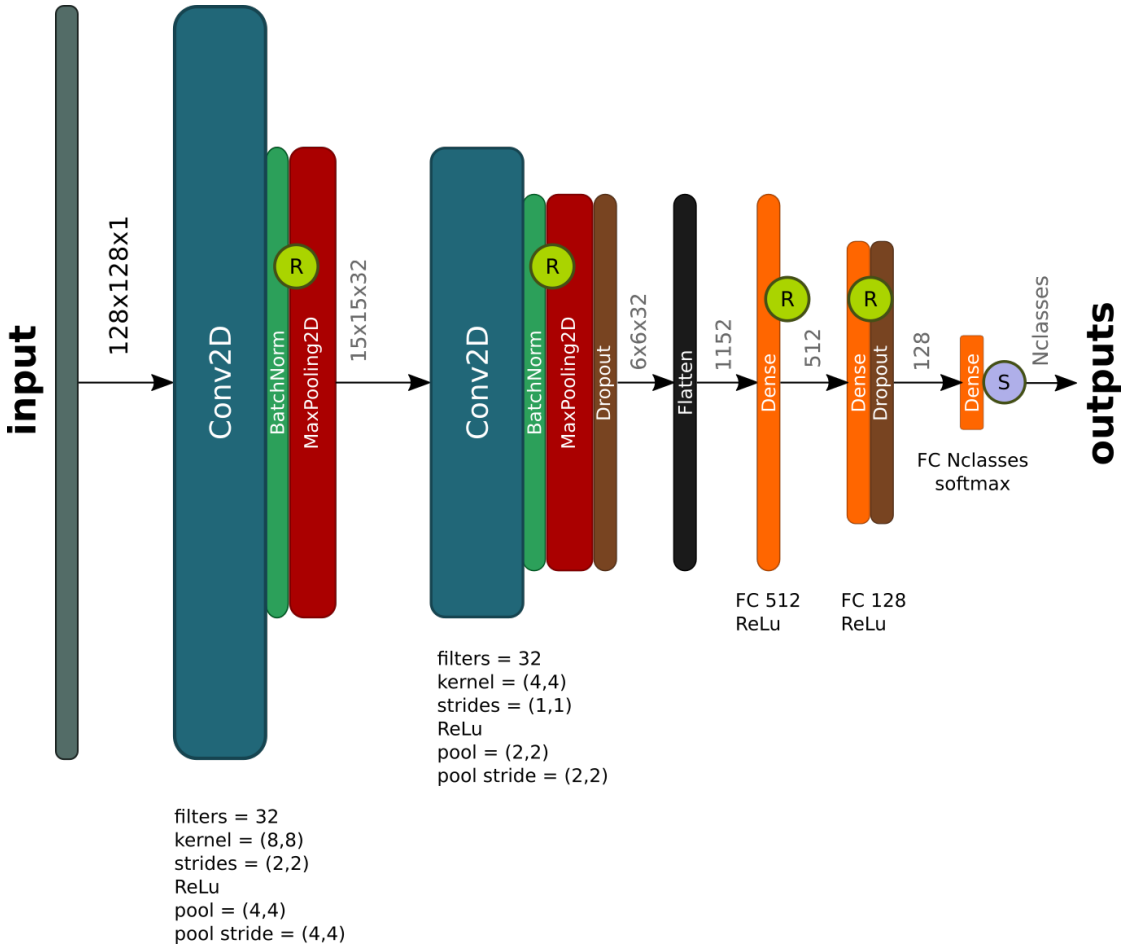


Figura 6.9: Modelo **granadero**.

hará una breve descripción de la arquitectura, de la motivación inicial y de algunos parámetros particulares de cada modelo, mientras que los resultados de desempeño se mostrarán recién en el capítulo 8. Los nombres que reciben los modelos son arbitrarios y no están relacionados con modelos preexistentes.

6.9.1. Granadero

Este modelo sigue la arquitectura convencional de una CNN. Es un modelo simple, con pocos parámetros, buscando tener pocas capas y pocos pesos a entrenar. La motivación principal fue tener una baja latencia de inferencia y que el modelo sea portable en plataformas con menores recursos computacionales, con un desempeño aceptable para la aplicación. Su diagrama se muestra en la figura 6.9.

La etapa de *feature learning* está conformada por 2 capas convolucionales 2D, donde la primera presenta una parametrización para realizar una fuerte reducción de la dimensión espacial del tensor de entrada (mayormente debido al tipo de *pooling*), consiguiendo una salida de dimensión $15 \times 15 \times 32$, para así bajar significativamente la

cantidad de pesos de la red; mientras que la segunda capa es más parecida a las convencionales. La salida de esta etapa tiene dimensión $6 \times 6 \times 32$, por lo que se hizo una fuerte reducción espacial de 128 píxeles a 6, y con una cantidad de canales moderada a baja. La reducción respecto al tensor de entrada se da en un factor de 14,2. Notar que en ambas capas convolucionales se utiliza *Batch Normalization*, buscando disminuir el error de generalización; al igual que con la inclusión del *Dropout* en la segunda capa.

La etapa de *classification* toma como entrada un tensor de 1152 elementos y está conformada por una red densa de 2 capas ocultas, donde la primera es de tamaño moderado (512 neuronas) y la segunda de un tamaño moderado a chico (128 neuronas). Se aplica *Dropout* en la segunda capa oculta para disminuir el error de generalización. También se utiliza una regularización de los kernels de las capas ocultas del tipo *Norma L2* (con un factor de regularización de 0,1).

Este modelo, para una salida de 6 elementos (mapeo 4, por ejemplo), tiene 675 398 parámetros entrenables (pesos) y 128 no entrenables (debido a las capas de *Batch Normalization*), donde 18 624 (2,76 %) pertencen a la etapa de *feature learning* y 656 774 (97,24 %) pertenecen a la etapa de *classification*.

Se evaluó un modelo denominado **garganta roja** con la misma arquitectura que granadero pero sin las capas de regularización. En realidad el modelo *granadero* es una evolución de *garganta roja*, incorporando la regularización para reducir el error de generalización. Algunos resultados comparativos pueden encontrarse en la sección 8.1.

6.9.2. Pollito

Este modelo está entre los modelos de mayor capacidad evaluados. Es un modelo más grande pero con una arquitectura convencional, con una cantidad mucho mayor de pesos a entrenar. La motivación de este modelo fue alcanzar un mejor desempeño en la clasificación en detrimento de la latencia y una exigencia de mayor capacidad computacional de la plataforma en donde se vaya a portar. En la figura 6.10 se muestra la arquitectura de este modelo. Notar que, si bien fue denominado como un modelo grande, sigue teniendo un tamaño conservador frente a los modelos convencionales de CNN utilizados para la clasificación de imágenes convencionales.

La etapa de *feature learning* está conformada por 4 capas convolucionales 2D, donde se va realizando una disminución progresiva de la dimensión espacial de los tensores y un incremento progresivo de la cantidad de canales. Los parámetros de las capas convolucionales y del pooling se ajustan a los estándares observados en muchos de los modelos CNN para clasificación de imágenes. La salida de esta etapa tiene dimensión $5 \times 5 \times 128$, por lo que se hizo una fuerte reducción espacial de 128 píxeles a 5, pero con una cantidad de canales alta (128) buscando capturar una gran cantidad de carac-

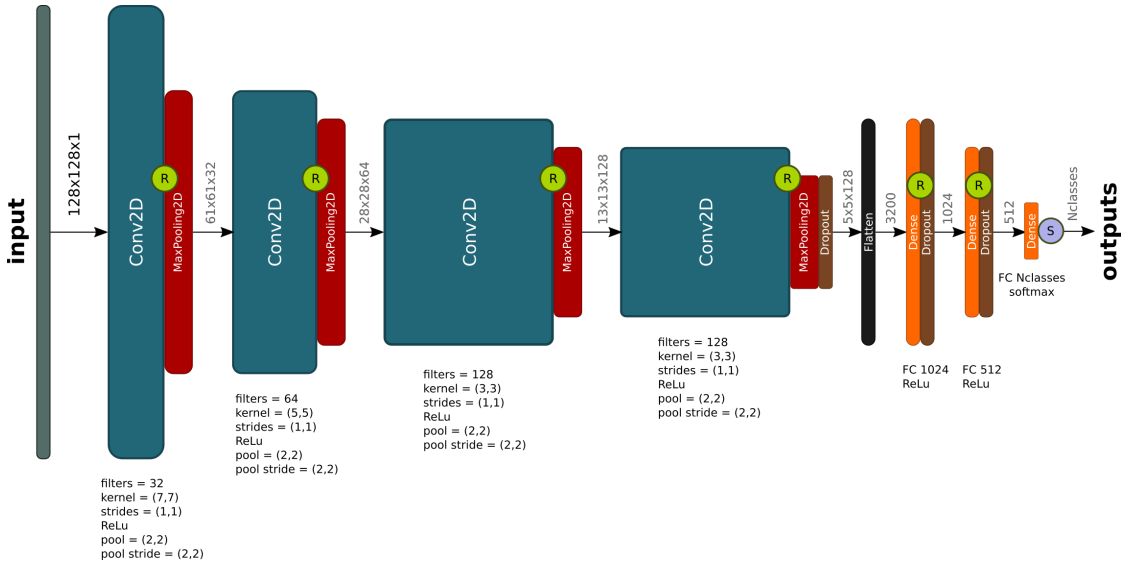


Figura 6.10: Modelo **pollito**.

terísticas. La reducción respecto al tensor de entrada se da en un factor de 5,12, mucho menor a la conseguida con el modelo *granadero*. La estrategia de regularización dentro de esta etapa se limita al uso de Dropout en la última capa convolucional.

La etapa de *classification* toma como entrada un tensor de 3200 elementos y está conformada por una red densa de 2 capas ocultas, donde la primera es de tamaño grande (1024 neuronas) y la segunda de un tamaño moderado (512 neuronas). Se aplica *Dropout* en ambas capas ocultas para disminuir el error de generalización.

Este modelo, para una salida de 6 elementos (mapeo 4, por ejemplo), tiene 4 080 006 parámetros entrenables (pesos), donde 274 304 (6,7 %) pertencen a la etapa de *feature learning* y 3 805 702 (93,3 %) pertenecen a la etapa de *classification*. Notar que este modelo tiene 6 veces más parámetros que el modelo *granadero*.

También se evaluaron variantes de este modelo. La variante **gallito** incorpora regularizaciones como *Batch Normalization* y *Kernel Regularization*, buscando reducir el error de generalización. La variante **pollex** hace una reducción fuerte del tamaño de las redes densas, que es donde se concentra la mayor cantidad de pesos del modelo. Este último modelo utiliza dos capas densas de 128 neuronas, lo que reduce la cantidad de pesos de la etapa de *classification* a 427 014 (una reducción de 8,9 veces), dejando la cantidad de pesos total en 701 318, casi igual a la del modelo *granadero*.

6.9.3. Grillo

Este modelo es una red neuronal convencional de capas densas. El modelo consta de dos capas ocultas y se evaluó con el propósito de comparar comportamiento y resultado frente a los modelos convolucionales. Este tipo de modelo es muy común dentro de los

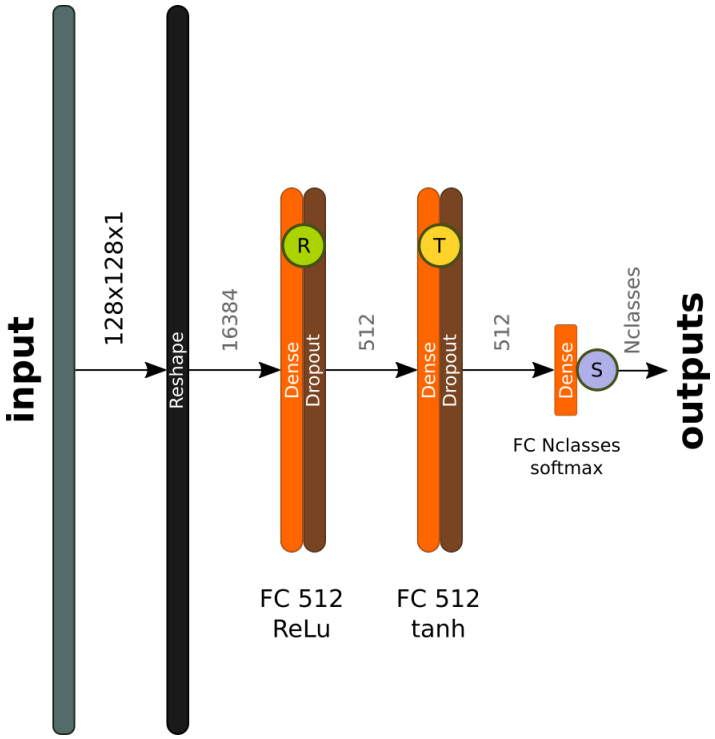


Figura 6.11: Modelo **grillo**.

clasificadores convencionales de ML. En este caso se construyó el modelo con capas de tamaño moderado (512 neuronas), donde la primera utiliza la activación *ReLU* y la segunda la activación *tanh*. La capa de salida es idéntica al resto de los modelos. La estrategia de regularización es aplicar Dropout de 50 % a ambas capas ocultas. Por la característica de las capas de este tipo de red, la entrada debe convertirse en un vector unidimensional de 16 384 elementos.

Este modelo, para una salida de 6 elementos (mapeo 4, por ejemplo), tiene 8 654 854 parámetros entrenables (pesos). Notar que este modelo tiene, aproximadamente, 2 veces más parámetros que el modelo *pollito*; cuando este último modelo tiene en su etapa de clasificación una red densa más grande. Este aumento en la cantidad de parámetros se debe a que el tensor de entrada no ha sido reducido en dimensiones como sí ocurre en las redes convolucionales.

Capítulo 7

Entrenamiento

Con los datasets definidos y los modelos propuestos, se pasa a la etapa de entrenamiento de dichos modelos usando la partición de entrenamiento del dataset seleccionado y luego a la evaluación del desempeño obtenido. La ejecución de estas tareas la realiza la *Training Chain* presentada en 3.3.

En este capítulo se hará una introducción a los conceptos involucrados en el proceso de entrenamiento supervisado y evaluación de desempeño de cada modelo, tales como el pre-procesamiento de las muestras, función de costo, optimización de los parámetros, métricas para la evaluación del desempeño y pos-procesamiento de los resultados de inferencias realizadas por el clasificador. Acompañado a esto se mostrarán los parámetros principales de configuración de esta etapa.

Los resultados para las distintas combinaciones de datasets y modelos serán presentados recién en el capítulo 8.

7.1. Método de entrenamiento

En **aprendizaje supervisado** todas las muestras de entrenamiento se encuentran etiquetadas, o sea, se conoce el valor que debería entregar el modelo de clasificación en su salida. En proceso de entrenamiento supervisado consiste, entonces, en ajustar los pesos del modelo para minimizar el error obtenido en la salida (diferencia entre el valor esperado y el valor obtenido como vector de salida). Un paso previo al entrenamiento propiamente dicho puede ser la estandarización o normalización de las muestras (7.3), para hacer un ajuste particular a las muestras, en función de las necesidades del modelo, que no sean satisfechas por el procesamiento realizado durante la conformación del dataset.

El primer paso en el entrenamiento del modelo es su inicialización, o bien, la **inicialización de los pesos** (7.2). Los pesos pueden inicializarse aleatoriamente o bien usando los de un modelo igual previamente entrenado (con otro dataset), lo que se conoce como *Transfer Learning* (transferencia de aprendizaje). Una vez realizada la inicialización de los pesos, se puede realizar una inferencia (también denominada feed-forward, predicción o clasificación), realizando un mapeo del espacio de entrada $f : \mathbb{R}^{N_h \times N_w \times N_{ch}} \rightarrow \mathbb{R}^{N_{classes}}$, tal que la salida de nuestro modelo puede expresarse como:

$$\hat{\boldsymbol{y}}^{(i)} = f(\boldsymbol{x}^{(i)}; \boldsymbol{W}) \tag{7.1}$$

entendiéndose a $\hat{\boldsymbol{y}}^{(i)}$ como el tensor de salida del modelo que contiene los resultados de la inferencia realizada para la muestra (tensor) $\boldsymbol{x}^{(i)} = \mathbf{X}_{:,:,:,i}$, cuando el modelo que define la transformación f está parametrizado con los pesos $\boldsymbol{W}$. Como es conocido el valor $\boldsymbol{y}^{(i)}$ que debería resultar de la inferencia, el error δ para la muestra i se puede calcular como:

$$\delta_i = \boldsymbol{y}^{(i)} - \hat{\boldsymbol{y}}^{(i)} \tag{7.2}$$

Es de esperar, que con pesos inicializados aleatoriamente el error obtenido sea grande para cualquiera de las muestras del dataset; no así necesariamente si se hizo *transfer learning*. Idealmente, el proceso de entrenamiento debería llevar a que $\delta_i = 0$ para todas las muestras del dataset.

A partir del error, que en esta aplicación representa un vector cuya dimensión es la cantidad de clases, se debe obtener un valor escalar al que denominaremos **Loss** (costo), que represente una medida del error global considerando el error en cada componente. Esa transformación se realiza utilizando una función determinada que se denomina **Loss Function** (función de costo), ver 7.4. La función de costo $\mathfrak{L}$ entregará un valor

de costo L_i para la inferencia realizada usando la muestra i, que dependerá del error de inferencia (o de manera más general del valor verdadero y predicho) y de parámetros adicionales dependiendo de la función elegida:

$$L_i = \mathfrak{L}(\boldsymbol{y}^{(i)}, \hat{\boldsymbol{y}}^{(i)}, ...) \tag{7.3}$$

El costo se calcula sobre un conjunto de muestras, generalmente sobre todas las muestras; por lo que tendremos un valor de costo para las muestras de entrenamiento y un valor de costo para las muestras de evaluación. La función de costo puede ampliarse con un término de **regularización**, que en general es función de los pesos del modelo. Tenemos entonces que el costo L calculado para un conjunto de N muestras puede escribirse como [76]:

$$L = \underbrace{\frac{1}{N}\sum_{i} L_i}_{\text{data loss}} + \underbrace{\lambda R(\boldsymbol{W})}_{\text{regularization loss}} \tag{7.4}$$

donde $R(\cdot)$ es la función de regularización y λ es el factor de regularización utilizado para ajustar la ponderación de la regularización en el costo.

En esta instancia, con la definición e inicialización del modelo y la definición de nuestra función de costo, se ha conseguido reducir un conjunto de muestras $\mathbb{X}$ y sus correspondientes etiquetas $\mathbb{Y}$, a un valor escalar tal como:

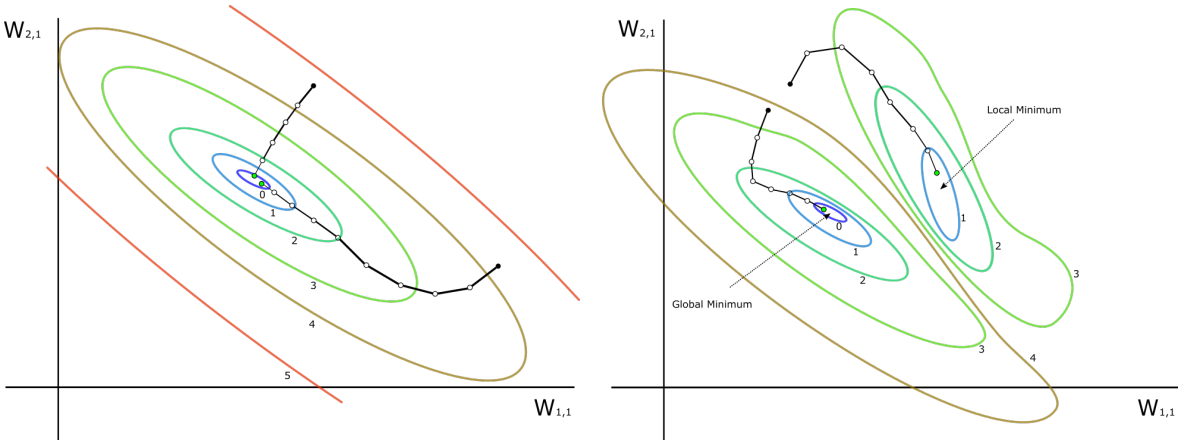
$$L = \mathfrak{L} \circ f(\mathbb{X}, \mathbb{Y}; \boldsymbol{W}, \Theta) = \mathfrak{L}_f(\mathbb{X}, \mathbb{Y}; \boldsymbol{W}, \Theta) \tag{7.5}$$

donde Θ expresa todos los hiper-parámetros del modelo.

Como este costo L es proporcional al error medio de las inferencias realizadas para el conjunto de muestras, el entrenamiento será entonces un proceso de optimización, que se realiza como la **minimización de la función de costo** ajustando los pesos $\boldsymbol{W}$ del modelo, o sea:

$$\arg\min_{\boldsymbol{W}} \mathfrak{L}_f(\mathbb{X}_{\text{train}}, \mathbb{Y}_{\text{train}}; \boldsymbol{W}, \Theta) \tag{7.6}$$

Según 7.5 podemos ver al costo como una función del espacio de los pesos $\boldsymbol{W}$ del modelo. Entonces, los valores de L definen una hiper-superficie con soporte en el espacio de estos pesos, como puede observarse en los casos de la figura 7.1. Hay casos en donde la función de costo es convexa, presentando un sólo mínimo (ver figura 7.1a), como por ejemplo el caso de un clasificador lineal en donde la función de costo es el Mean Squared Error (Error Cuadrático Medio) (MSE), donde la superficie definida en este



(a) único mínimo global. **(b)** mínimo global y mínimo local.

Figura 7.1: Función de costo graficada por niveles en el espacio de dos pesos. En ambas figuras se muestran la optimización (búsqueda del mínimo) para dos casos con valores iniciales distintos de los pesos. En (a) se muestra el caso en donde la función de costo presenta un único mínimo global. En (b) se muestra el caso en donde hay un mínimo global y un mínimo local; dependiendo de la inicialización de los pesos y del algoritmo de optimización, los pesos finales pueden terminar en mínimos diferentes.

caso es una hiper-parábola. En el caso de las redes neuronales, por el tipo de mapeo (no lineal) que realiza el modelo, la superficie generada puede presentar más de un mínimo; de esta manera, encontrar el mínimo global resulta más difícil (ver figura 7.1b).

El proceso de minimización [76, 77, 83] de la función de costo es llevada a cabo por el **Optimizer** (optimizador, ver 7.5), el cual define una estrategia para encontrar el mínimo global. La estrategia naive sería explorar el espacio de los pesos y quedarnos con el conjunto de pesos que resulta en el mínimo costo, pero dada la gran cantidad de parámetros esto resulta impracticable. En la práctica, la estrategia más utilizada se basa en el **descenso por gradiente** [84]. Este método parte de un valor inicial de $\boldsymbol{W}$ y calcula el gradiente de la función de costo para ese punto, donde el gradiente es un vector que indica la dirección en la que la función cambia más rápidamente, y cuyo módulo representa la tasa de cambio de la función en esa dirección. Al gradiente [85] de la función de costo lo escribimos como:

$$\nabla_W \mathfrak{L}_f = \frac{\partial \mathfrak{L}_f(W)}{\partial \boldsymbol{W}} \tag{7.7}$$

A partir del gradiente calculado, el optimizador puede modificar cada uno de los pesos del modelo para asegurar que habrá una disminución del costo con los nuevos valores de los pesos. Se puede pensar en la analogía de estar parado en una montaña y dar un paso en la dirección de máxima pendiente negativa. La magnitud del cambio en los pesos es parte de la estrategia que sigue el algoritmo del optimizador y que determinará la velocidad de convergencia a un mínimo de la función, como así también la estabilidad del algoritmo. En la figura 7.1 se pueden ver ejemplos de este proceso.

En el límite, siguiendo este método, se asegura encontrar un mínimo local de la función (siempre que la magnitud de la actualización no comprometa la estabilidad). Podemos expresar la actualización de los pesos [84] como:

$$\boldsymbol{W}_{n+1} = \boldsymbol{W}_n - \gamma \nabla_W \mathfrak{L}_f(\boldsymbol{W}_n) \tag{7.8}$$

o bien, usando la notación de asignación:

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \gamma \nabla_W \mathfrak{L}_f(\boldsymbol{W}) \tag{7.9}$$

donde el factor γ es un factor establecido por el optimizador (o bien definido por el usuario como un parámetro), que puede ser constante o adaptivo. En muchos casos recibe el nombre de **learning rate** (tasa de aprendizaje), que es uno de los hiper-parámetros más importantes a elegir para nuestro modelo.

Realizar el cálculo del gradiente usando una función analítica del modelo puede ser relativamente sencillo, pero demanda una capacidad de cómputo muy grande, y más aún cuando los modelos a entrenar poseen una gran cantidad de pesos. En la práctica, se utiliza un algoritmo denominado **Back-Propagation** [86], que permite realizar un ahorro significativo de cómputos haciendo fluir la información del costo hacia atrás usando el modelo como un grafo computacional. De manera resumida se puede decir que este algoritmo, al conocer la salida (costo), la función de costo (se utiliza la derivada de dicha función) y los valores de entrada a dicha función; puede calcular la sensibilidad de cada entrada al valor de salida, por lo tanto la componente del gradiente correspondiente a esos valores. Siguiendo el flujo hacia atrás en la red, o grafo computacional, los valores obtenidos se propagan una capa más en dicho sentido, que en este caso sería la salida del modelo; luego se propagará a través de la última capa oculta; y así sucesivamente hasta llegar a la entrada del modelo. Luego de realizado dicho flujo computacional, se dispone del gradiente de la función de costo en función de todos los parámetros que intervinieron en el mapeo de un dato de entrada a un valor de costo. Este algoritmo hace uso de las derivadas de las funciones por las cuáles fluyeron los datos durante la inferencia o forward-pass, más precisamente utilizando la regla de la cadena; por lo que, derivadas sencillas implican un cálculo más liviano, por lo tanto un entrenamiento más rápido. Para más detalle del algoritmo se puede consultar: [1, 76, 77].

No hay que confundir este algoritmo con el que utiliza el optimizador para ajustar los pesos del modelo, ya que el algoritmo de Back-Propagation se utiliza solamente para el cálculo del gradiente en función del modelo y de un valor obtenido como costo en la salida del modelo. De manera general, este algoritmo se puede aplicar a cualquier

función expresada como un grafo computacional, por lo que su uso no se limita a ML. Todo este proceso de obtención del gradiente se realiza muestra a muestra, ya que por cada muestra se obtiene un valor de costo diferente. No en todos los casos se ajustan los pesos por cada gradiente calculado, sino que se realiza un promediado de los gradientes obtenidos para un conjunto reducido de muestras, al que se definió como *mini-batch*, o **batch**. Entonces, la cantidad de muestras de cada batch es un hiper-parámetro de la *Train-Chain*, que tiene impacto en la dinámica del descenso por gradiente de la función de costo. El dataset de entrenamiento se particiona aleatoriamente en batches de acuerdo al tamaño definido, se calcula el gradiente medio por cada batch y por cada gradiente medio se van ajustando los pesos. Todas esas etapas se repiten hasta agotar la cantidad de muestras del dataset de entrenamiento, denominando a ese proceso **época**. El entrenamiento consiste en repetir el proceso durante una determinada cantidad de épocas hasta alcanzar el desempeño deseado, mientras sea posible. Los métodos que utilizan todo el dataset para recién hacer la actualización se denominan *determinísticos*, mientras los que usan una sola muestra o un conjunto reducido (mini-batch) se suelen denominar *estocásticos* u *online*.

Nota : El término *batch* cuando se usa como algoritmo de optimización se refiere a aquél algoritmo que hace uso del dataset completo de entrenamiento para recién ajustar los pesos del modelo (métodos de gradiente determinístico), mientras que es común usar el mismo término para referirse a un conjunto de muestras de dicho dataset cuando es particionado para el entrenamiento, donde el término correcto, estrictamente hablando, sería *mini-batch*; usándose generalmente el término *batch size* para referirse a la cantidad de muestras de dicho mini-batch.

7.2. Inicialización de los pesos

En la práctica, la inicialización de los pesos del modelo se realiza de manera aleatoria, generalmente utilizando distribuciones normal o uniforme, en rangos muy cercanos a cero (varianzas pequeñas). La principal razón detrás de realizar una inicialización de este tipo es realizar una *ruptura de simetría*, evitando que la actualización de los pesos de la red no sean similares durante la evolución del entrenamiento, lo que mejora la convergencia del modelo durante el entrenamiento [76].

Uno de los problemas con la inicialización aleatoria de los pesos, es que la varianza de la distribución de los valores en la salida de una neurona, crece con la cantidad de entradas de la misma. Resulta conveniente, entonces, ajustar la varianza de la distribución con la que se inicializan los pesos de una neurona en función de la cantidad de entradas. Una regla heurística que suele aplicarse es la expresada en la ecuación 7.10:

$$\boldsymbol{W} \sim \mathcal{N}(0, 1/n) \sim \frac{1}{\sqrt{n}}\mathcal{N}(0, 1) \tag{7.10}$$

donde n es el número de entradas de la neurona en cuestión. De esta manera se asegura que la salidas de las distintas neuronas que se inicializan con este criterio, tienen una varianza de 1. Si bien se especificó una distribución normal, esta podría ser una distribución uniforme (también usada ampliamente en la práctica), o cualquier otra con propiedades similares.

Existen algunos otros métodos heurísticos que funcionan igualmente bien, y que se ajustan mejor de acuerdo al tipo de arquitectura del modelo. Una de las alternativas fue propuesta por *He* [87], donde dicho método se ajusta mejor cuando el modelo utiliza neuronas del tipo ReLU, proponiendo que la varianza resultante debe ser:

$$\mathrm{Var}(\boldsymbol{W}) = \frac{2}{n} \tag{7.11}$$

También existe la variante del método para una distribución uniforme, donde:

$$\boldsymbol{W} \sim \mathcal{U}[-\mathrm{limit}, \mathrm{limit}] \tag{7.12}$$

$$\mathrm{limit} = \sqrt{\frac{6}{n}} \tag{7.13}$$

Otro método muy utilizado es el propuesto por *Glorot* [88] en donde también incorpora la cantidad de conexiones entrantes a una capa n_{in} (*fan-in*) como lo presentado anteriormente; pero que también considera la cantidad de conexiones saliente n_{out} (*fan-out*). Proponiendo que la varianza para una distribución normal sea:

$$\mathrm{Var}(\boldsymbol{W}) = \frac{2}{n_{in} + n_{out}} \tag{7.14}$$

Mientras que si se elige una distribución uniforme según 7.12:

$$\mathrm{limit} = \sqrt{\frac{6}{n_{in} + n_{out}}} \tag{7.15}$$

Estas variantes de inicialización suelen aplicarse solamente a los pesos que multiplican las entradas a cada neurona, mientras que los pesos que se denominaron *bias* suelen inicializarse en 0. En algunos casos, más que nada cuando se trabaja con neuronas ReLU, se puede optar por inicializarlos con un valor constante pequeño (respecto a la desviación estándar de los valores esperados en la entrada de cada capa), por ejemplo 0,01, para pre-activar a este tipo de neuronas.

En los modelos propuestos en este trabajo se inicializarán los pesos siguiendo el método *Glorot Uniforme*, a menos que se indique explícitamente algún otro método. Para el caso de los *bias*, su inicialización por defecto será en 0.

7.3. Estandarización de las muestras

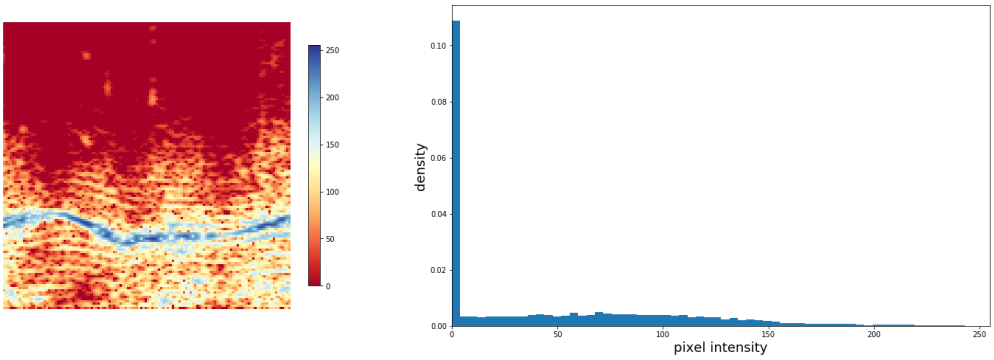
Una tarea de pre-procesamiento recomendada antes de alimentar con datos al modelo, es realizar una *estandarización* de los mismos. Este procesamiento consiste básicamente en suprimir la media y luego escalar los datos de manera de normalizar su varianza. De esta manera, se adecuan las distribuciones de los datos a los rangos de mayor variación de las funciones de activación. Las variantes en el proceso de estandarización están relacionadas al valor de media y varianza que se utilizan para normalizar cada componente de la muestra de entrada [76].

En las redes neuronales convencionales, la tarea de clasificación se realiza sobre muestras que, generalmente, representan un vector de *features* heterogéneas; o bien, dicho de otra manera, cada componente del vector de entrada tiene una distribución distinta al resto. En estos casos lo conveniente es estandarizar cada componente por separado, o sea, utilizando sólo la distribución de los datos de esa componente en el dataset de entrenamiento. Aplicando ese proceso a cada componente, tendremos muestras donde cada componente tendrá una distribución similar, con media cero y varianza unitaria. De esta manera:

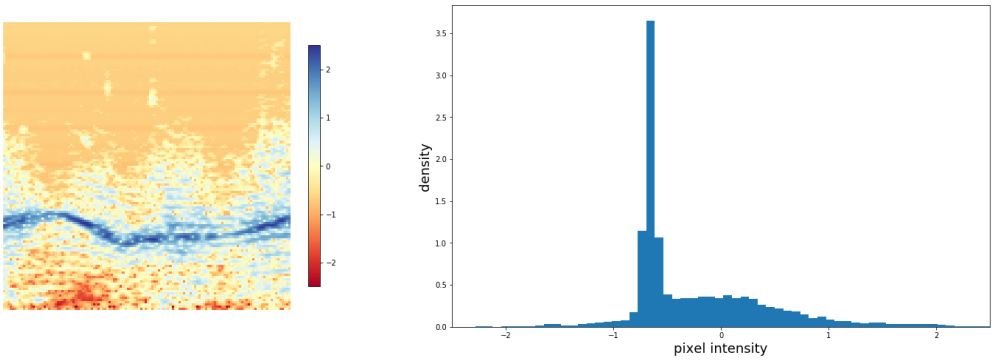
$$\tilde{\boldsymbol{X}}_i^{(k)} = \frac{\boldsymbol{X}_i^{(k)} - \mu_{\boldsymbol{X}_i}}{\sigma_{\boldsymbol{X}_i}} \tag{7.16}$$

donde $\tilde{\boldsymbol{X}}_i^{(k)}$ es la componente i de la *k-ésima* muestra del dataset $\mathbb{X}$, siendo $\mu_{\boldsymbol{X}_i}$ y $\sigma_{\boldsymbol{X}_i}$ la media y desviación estándar de la componente i de las muestras del dataset $\mathbb{X}_{train}$.

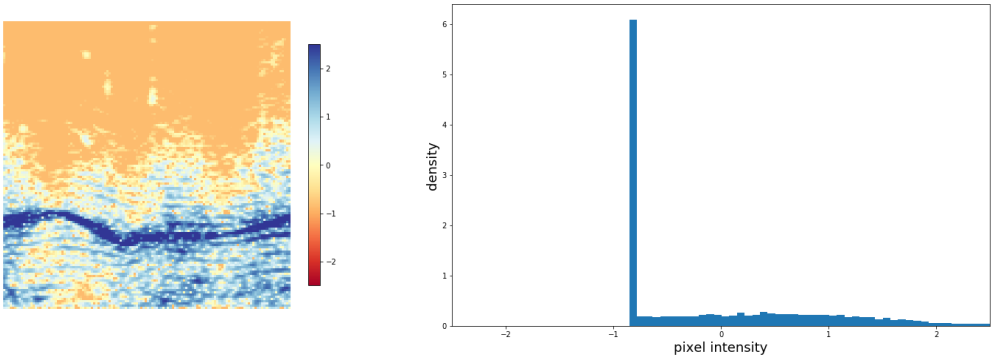
En las redes convolucionales realizar una estandarización independiente por cada componente del tensor de entrada (píxel de una imagen) puede llevar a inducir distorsiones espaciales en cada muestra, por lo que se pueden "romper"patrones espaciales que se buscan reconocer en el proceso de convolución. Por otro lado, puede tener la ventaja de suprimir patrones repetidos a lo largo de las muestras de entrenamiento, que en la aplicación presentada en este trabajo puede ayudar a reducir interferencias o niveles de ruido presentes en todas las muestras, que se deben principalmente a fuentes internas al radar. Otro aspecto a considerar es que en la conformación del dataset, durante la etapa de Pre-Procesamiento de las muestras (5.5) se realizó un mapeo del rango dinámico de los valores del espectrograma (o lo que corresponda de acuerdo al método elegido) al rango de valores definido por el tipo de dato usado para cada com-



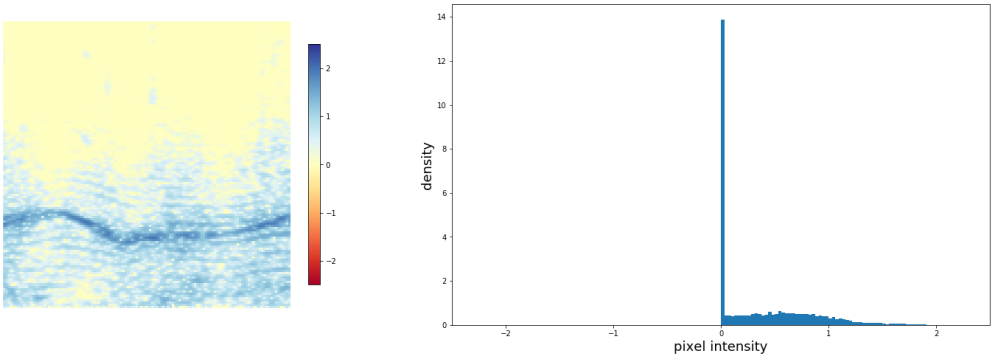
(a) original



(b) mean_var_pixel



(c) mean_var_sample



(d) minmax (bias = 1, scale = 1)

Figura 7.2: Resultados de algunos métodos de estandarización de una muestra: *mean_var_pixel* (b), *mean_var_sample* (c) y *minmax* (d), en comparación con la muestra original (a). En cada imagen se muestra el espectrograma y el histograma de las intensidades de cada píxel.

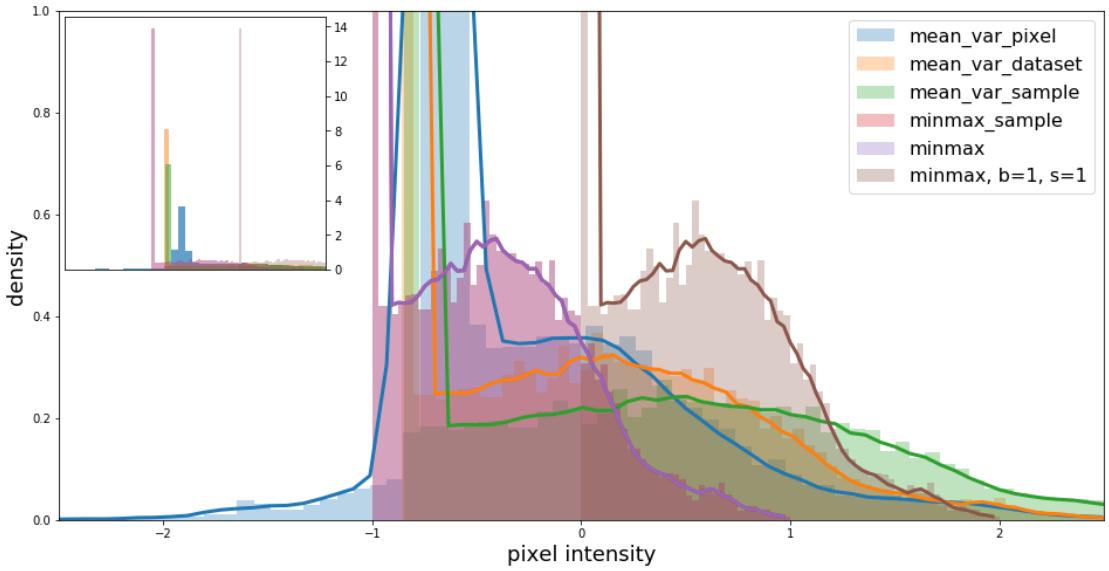


Figura 7.3: Histogramas superpuestos de las intensidades de los píxeles de la imagen resultante al aplicar distintos métodos de estandarización.

ponentes, que en este caso era UINT8. De esta manera los valores de las muestras se encuentran distribuidos en el rango de $[0, 255]$. Una alternativa al método de estandarización mostrado anteriormente es simplemente realizar un re-escalado de los datos para adecuar el rango de $[0, 255]$ a $[-1, 1]$, o $[0, 1]$.

En general, las redes convolucionales no presentan una gran sensibilidad al tipo de estandarización usada, ya que durante el entrenamiento adecua los valores de los filtros y bias dependiendo de las distribuciones de los datos. Por lo que la elección del método de estandarización no suele ser crítica, pero puede tener algún beneficio en la velocidad de convergencia. En la *Train Chain* se han implementado diversos métodos de estandarización, en donde el método deseado se selecciona mediante la parametrización de dicha cadena:

```
{
"standarization_method": "mean_var_sample",
"standarization_bias": 0.0,
"standarization_scale": 1.0,
}
```

Cualquiera de los métodos de estandarización realizan:

$$\tilde{\boldsymbol{X}}_i^{(k)} = \left(\frac{\boldsymbol{X}_i^{(k)} - \mu}{\sigma} + b \right) \cdot s \tag{7.17}$$

donde μ y σ serán valores determinados por el método de estandarización elegido y b y s serán valores de *bias* y *scale* definidos arbitrariamente por los parámetros *standarization_bias* y *standarization_scale* respectivamente.

A continuación una breve descripción de los métodos implementados en la cadena:

- **mean_var_pixel**: μ y σ se corresponden a la media y desviación estandar por cada componente a lo largo del dataset de entrenamiento, tal como 7.16.
- **mean_var_dataset**: μ y σ se corresponden a la media y desviación de todas las componentes y todas las muestras del dataset de entrenamiento, por lo que ambas son un valor escalar que se aplica a todas las componentes por igual.
- **mean_var_sample**: μ y σ se corresponden a la media y desviación de todas las componentes pero calculadas muestra a muestra. Esto quiere decir que a cada muestra de entrada se la estandariza con su media y desviación estándar, y no con la del dataset.
- **minmax_sample**: μ se corresponde al punto medio entre el valor mínimo y máximo de la muestra (bias); y σ a la diferencia entre dichos valores (span) dividida por 2. Esto mapea los valores de las componentes al rango $[-1, 1]$.
- **minmax**: μ se corresponde al punto medio entre el valor mínimo y máximo del tipo de dato elegido para cada componente (bias); y σ a la diferencia entre dichos valores (span) dividida por 2. Esto mapea los valores de las componentes al rango $[-1, 1]$.

En la figura 7.2 se muestra una comparación entre algunos de los métodos de estandarización para una muestra tomada al azar. Notar que en la muestra original 7.2a posee una gran cantidad de píxeles con intensidades bajas debido a que representan el piso de ruido del espectrograma, mientras que la información relacionada a la señal micro-Doppler propiamente dicha está representada por una cantidad mucho menor de píxeles. Cualquiera de los métodos, a excepción del método *mean_var_pixel*, mantiene a grandes rasgos la forma de la distribución del histograma de intensidades, pero variando el rango en el que quedan mapeados los valores de intensidad, como en 7.2c y 7.2d. En la figura 7.3 se muestra una comparación de todos los métodos para la misma muestra.

Notar que en el caso de *mean_var_pixel*, como se muestra en 7.2b, aparece una deformación del histograma en función de las distribución de las componentes en las diferentes clases que dispone el dataset. La imagen muestra un realce de componentes que no están presentes en la muestra (interferencias de frecuencias constantes, líneas horizontales) y de la media de los valores dependiendo del rango de frecuencias (el piso de ruido a bajas frecuencias tiene mayor intensidad que en las altas). Puede verse en el histograma que la intensidad del piso de ruido en general ahora no ocupa los valores de intensidad más baja, lo que puede ser interesante al llevar más cerca de cero dichos píxeles. Esta última cualidad (intensidad del piso de ruido en el cero del mapeo de intensidades) también puede ajustarse usando el valor de *bias* arbitrario, donde un

ejemplo se muestra en la figura 7.2d para el método *minmax*.

7.4. Función de costo

La definición de la función de costo utilizada para el entrenamiento del modelo, tal como se explicó en la sección 7.1, depende del tipo de problema que se quiera resolver. Existen diversas funciones que se pueden usar, donde muchas de ellas son las que se utilizan para otros modelos paramétricos, como los modelos lineales.

En muchos casos, inclusive en el tipo de problema que se quiere resolver en este trabajo, nuestro modelo paramétrico define una distribución $p(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{W})$ y simplemente se utiliza el principio de máxima similitud. Esto significa que se utiliza como función de costo la *cross-entropy* (entropía cruzada) entre los datos del dataset de entrenamiento y las predicciones que realiza el modelo [1]. La entropía-cruzada entre la probabilidad real p de una muestra respecto a la probabilidad de la predicción q realizada por el modelo se escribe como:

$$H(p, q) = -\sum_{\boldsymbol{x} \in \mathbb{X}} p(\boldsymbol{x}) \log q(\boldsymbol{x}) \tag{7.18}$$

Nuestro problema trata de una clasificación de múltiples clases, por lo que cada muestra tiene una etiqueta $\boldsymbol{y}^{(i)}$ que contiene la probabilidad de cada clase para la i-ésima muestra del dataset. Este vector se codifica como **one-hot**, es decir, tiene un 1 en la componente que pertenece a su clase, y 0 en el resto de las componentes, ya que no hay muestras que pertenezcan a más de una clase a la vez. Por otro lado, nuestro modelo entrega un vector $\hat{\boldsymbol{y}}^{(i)}$ con las probabilidades inferidas usando la función *Softmax* según 6.7, que en el caso óptimo $\hat{\boldsymbol{y}}^{(i)} = \boldsymbol{y}^{(i)}$; pero que en general al ir entrenando el modelo, la componente de la clase correcta se debería ir aproximando a 1 y el resto a 0. Esta función de costo se denomina **cross-entropy loss** [89] y es ampliamente usada en este tipo de aplicaciones, más aún cuando la función de activación de la salida del modelo es la función *Softmax*; cuando se utilizan múltiples salidas (clases), la función recibe el nombre de **categorical cross-entropy loss** . De esa manera, podemos escribir que el costo para una muestra dada lo podemos calcular como:

$$L_i = -\sum_k p(\boldsymbol{x}_k^{(i)}) \log q(\boldsymbol{x}_k^{(i)}) = -\sum_k \boldsymbol{y}_k^{(i)} \log \hat{\boldsymbol{y}}_k^{(i)} \tag{7.19}$$

donde k recorre las componentes asociadas a las clases. Como se está usando la codificación *one-hot*, todas las componentes de $\boldsymbol{y}^{(i)}$ correspondientes a la clase incorrecta son 0, por lo que los términos en la sumatoria se anulan. Si, a la vez, introducimos que

la salida está definida por la función *Softmax* tenemos:

$$L_i = -\log \hat{\boldsymbol{y}}_{k_c}^{(i)} = -\log \left(\frac{e^{f_{k_c}}}{\sum_{k=1}^{K} e^{f_k}} \right) \tag{7.20}$$

donde k_c es la componente correspondiente a la clase correcta. Se usa f_k para expresar el valor de salida del modelo para la componente k antes de aplicar la función *Softmax*. [76]. Notar que cuando la probabilidad entregada por el modelo es $\hat{\boldsymbol{y}}_{k_c}^{(i)} = 1$, el costo L_i es igual a 0 para esa muestra.

El costo L_i, corresponde al costo calculado para una muestra dada en una época en particular del entrenamiento del modelo y es el término L_i de la ecuación 7.4. El costo para el conjunto de todas las muestras se calcula como el promedio de los costos para cada una de las muestras (término *data loss* de la ecuación 7.4).

En el entorno de trabajo, por defecto, la función de costo se define junto al modelo durante su compilación y no en la *Train Chain*. Sin embargo, se puede forzar una función de costo para la sesión parametrizando la *Train Chain* con:

```
{
"overwrite_compilation": true,
"loss": "categorical_crossentropy"
}
```

Por defecto se utiliza la función de costo *Categorical Cross-Entropy* para los modelos propuestos, pero podrían usarse alternativas varias como: MSE, Mean Absolute Error (Error Absoluto Medio) (MAE), Mean Squared Logarithmic Error (Error Logarítmico Cuadrático Medio) (MSLE), Categorical Hinge, Kullback-Leibler Divergence (Divergencia de Kullback-Leibler) (KLD), entre otras.

7.4.1. Regularización

En este punto hemos definido la función de costo propiamente dicha, que se corresponde al término *data loss* de la ecuación 7.4. Sin embargo, como se expresa en la misma ecuación, puede incluirse un término de regularización destinado a penalizar el costo en función de los valores de los pesos del modelo (en general se usan los pesos multiplicativos y no los bias), limitando así la capacidad del modelo. Este término es opcional y suele usarse y ajustarse en función del sobre-ajuste (overfitting) que tenga el modelo, donde dicho ajuste se realiza heurísticamente mediante el parámetro λ. Existen diversas maneras de controlar la capacidad del modelo para prevenir el sobre-ajuste, pero se presentarán dos tipos de regularización que fueron usadas en la evaluación de

modelos.

El modo más común de regularización en modelo de redes neuronales es el denominado L^2 **regularization** (regularización L^2), también conocida como *weight decay* (decaimiento de los pesos) [1], o bien, *ridge regression* o *Tikhonov regularization*. Se implementa penalizando el costo en función de la magnitud cuadrática de los pesos del modelo. De esta manera, el término de regularización de la función de costo se escribe como:

$$R(\boldsymbol{W}) = \frac{1}{2}||\boldsymbol{W}||_2^2 \tag{7.21}$$

el término $1/2$ se agrega simplemente para que el gradiente de este término sea $\lambda \boldsymbol{W}$, recordando que el factor λ define la ponderación de la regularización en la función de costo total, 7.4. Notar que durante la actualización de los pesos por gradiente, esta regularización aporta con $-\lambda \boldsymbol{W}$, por lo que decae linealmente a cero, y de ahí el nombre de *weight decay*.

Este tipo de regularización puede interpretarse, intuitivamente, como una penalización mayor a pesos con valores grandes, dando preferencia sobre pesos difusos, cercanos al origen. De esta forma, también se induce a que una capa en particular tienda a usar todas las entradas y no algunas en particular (las correspondientes a pesos altos), [76]. Por el análisis mostrado en [1], esta regularización también genera una insensibilización de la función de costo a las direcciones del gradiente que no contribuyen mucho a la reducción de dicha función, a medida que transcurre el entrenamiento.

Nota : Recordar que el modelo *granadero* (ver 6.9.1) utilizaba la regularización L^2 en las capas densas de la etapa de clasificación, con $\lambda = 0{,}1$.

Otra de las opciones de regularización de la función de costo es la L^1 **regularization**, definida como:

$$R(\boldsymbol{W}) = ||\boldsymbol{W}||_1 = \sum_i |\boldsymbol{W}_i| \tag{7.22}$$

siendo básicamente la suma del valor absoluto de los pesos individuales del modelo. Notar que en este caso la contribución al gradiente es $-\lambda \operatorname{sign}(\boldsymbol{W})$, por lo que ahora dicha contribución no escala linealmente como en el caso de L^2. Esto lleva a que el comportamiento de esta regularización sea muy diferente de L^2, pudiendo dificultar encontrar soluciones algebraicas a aproximaciones cuadráticas de la función de costo, [1].

Uno de los comportamientos más relevantes observados al usar una regularización del tipo L^1 es que los pesos del modelos se vuelven *ralos (sparse)*, o sea que un conjunto de pesos alcanzarán valores muy próximos a cero, [1, 76]. Esta dispersión es extensamente usada como un mecanismo de selección de características (*feature selection*) relevantes; ya que los pesos que tienden a tener valores próximos a cero se suponen asociados a features irrelevantes, que también son denominadas *noisy features* (entradas ruidosas). De esta manera, las entradas correspondientes a esas features irrelevantes podrían descartarse sin afectar al desempeño.

La regularización de la función de costo en función de los pesos se define en la arquitectura del modelo, contando con la posibilidad de aplicarla y parametrizarla a nivel capa. De esa manera se pueden elegir en qué capas aplicar regularización, qué tipo de regularización y el factor λ para cada capa. Inclusive se pueden combinar las regularizaciones L^2 y L^1 en una misma capa, lo que se denomina **Elastic Net Regularization**.

7.5. Optimizador

El *optimizador* es el bloque de la *Train Chain* que realiza la actualización de los pesos durante el entrenamiento del modelo. Como se explicó en la sección 7.1, el optimizador realiza la actualización utilizando el gradiente calculado para un instante dado del entrenamiento. Si bien se presentó la actualización de esos pesos de manera general, como 7.8, existen diversos métodos para realizar dicha actualización. En este trabajo, se evaluaron los modelos con distintos métodos de optimización, pero luego se mostrarán sólo las combinaciones más relevantes. Algunos de los optimizadores utilizados durante el desarrollo del trabajo se describen brevemente a continuación.

El optimizador se define por defecto durante la compilación del modelo, pero esta configuración puede sobrescribirse como parámetros de la *Train Chain*, usando los parámetros *optimizer* y *optimizer_params*. A continuación se muestra un ejemplo:

```
{
"optimizer": "Adamax",
"optimizer_params": {
        "learning_rate": 0.001
        }
}
```

7.5.1. SGD

El método *Stochastic Gradient Descent* (SGD) y sus variantes, son uno de los métodos más utilizados para optimización en las aplicaciones de machine learning y deep learning. Este método realiza el ajuste de los pesos del modelo, tal como se expuso en 7.8, donde el hiperparámetro γ correspondiente, denominado *learning rate*, es el que define la proporción del ajuste en función del gradiente. [1, 90].

Cuando la actualización se realiza muestra a muestra, el método es lo que se denomina estocástico; sin embargo, lo usual es obtener un gradiente promedio considerando M_B muestras, donde definimos a ese conjunto como *mini-batch*. Rigurosamente, ese método ya no sería estocástico, pero es considerado la variante más común del método SGD, denominándose **Batch Gradient Descent** (BGD). Podemos actualizar la ecuación 7.8 incorporando el mini-batch $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(M_B)}\}$, obteniendo como regla de actualización a:

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \gamma \nabla_W \Big(\frac{1}{M_B} \sum_i \mathfrak{L}_f(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}; \boldsymbol{W}) \Big) \tag{7.23}$$

donde i recorre los elementos del mini-batch. Para simplificar la notación e incorporar la función de costo regularizada escribimos en adelante $L_i(\boldsymbol{W})$ para indicar el costo calculado para la muestra i usando los pesos $\boldsymbol{W}$ del modelo en ese instante.

Resulta conveniente que el learning-rate se vaya ajustando a medida que transcurren las épocas del aprendizaje, ya que en general son preferibles valores altos al principio, pero a medida que el aprendizaje transcurre, disminuir este valor ayuda a mejorar la velocidad de convergencia. En la práctica es común que este valor decaiga linealmente hasta la época τ:

$$\gamma_k = (1 - \alpha)\gamma_0 + \alpha\gamma_\tau \tag{7.24}$$

donde $\alpha = k/\tau$. Luego de la época τ es común dejar γ constante.

Durante la evaluación de los modelos, el learning rate se ajusta realizando una inspección del comportamiento del costo por época. De esta manera, si el costo disminuye muy lentamente se incrementa el learning rate hasta conseguir una velocidad de convergencia aceptable. Si el learning rate es muy grande, lo esperable es observar fuertes oscilaciones en la curva del costo vs. épocas. Cuando se modifica progresivamente el learning rate, lo usual es que el valor final no sea menor al 1 % del valor inicial. Si el learning rate es bajo, es esperable que la cantidad de épocas para alcanzar la convergencia sea elevada; e inclusive, si el learning rate es muy bajo, es probable que el costo quede estancado en un mínimo local.

En general el método Batch Gradient Descent (Descenso por Gradiente por Batch) (BGD) tiene una mejor tasa de convergencia que el método Stochastic Gradient Descent (Descenso por Gradiente Estocástico) (SGD), pero este último puede realizar un rápido progreso al inicio del entrenamiento en comparación con BGD, cuando se utilizan datasets muy grandes. A continuación se muestra un ejemplo de configuración de la *Train Chain* para este optimizador:

```
{
"optimizer": "SGD",
"optimizer_params": {
        "learning_rate": 0.1,
        },
"reduce_lr_enable": true,
"reduce_lr_monitor": "loss",
"reduce_lr_factor": 0.5,
"reduce_lr_patience": 15,
"reduce_lr_min_lr": 1e-4,
"reduce_lr_cooldown": 0
}
```

7.5.2. Momentum

Si bien el método SGD es muy utilizado para entrenar redes neuronales, dando buenos resultados, suele ser lento. El método de **momentum** (momento) está diseñado para acelerar el aprendizaje, especialmente en zonas del gradiente con una gran curvatura, pequeños gradientes pero consistentes, o bien gradientes ruidosos. Este algoritmo acumula un promedio deslizante de los gradientes pasados que decae exponencialmente y continua moviéndose en esa dirección.

Este método introduce una variable $\boldsymbol{v}$ que juega el rol de velocidad, tomando en cuenta dirección y velocidad a la que se mueven los pesos a través del espacio de parámetros. El nombre *momentum* deriva de la analogía con la física, en donde el gradiente equivaldría a una fuerza que mueve una partícula a través del espacio de parámetros, según las leyes del movimiento de Newton, donde el momento se define como la masa de la partícula por su velocidad. En lo que respecta a el aprendizaje, consideraremos esa masa como unidad, por lo que $\boldsymbol{v}$ termina representando el momento, [1]. La regla de actualización de los parámetros podemos escribirla como:

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \gamma \nabla_W \Big(\frac{1}{M_B} \sum_i L_i(\boldsymbol{W}) \Big), \tag{7.25}$$

$$\boldsymbol{W} \leftarrow \boldsymbol{W} + \boldsymbol{v} \tag{7.26}$$

donde α es el parámetro que pondera el efecto del momento sobre la actualización de los pesos solamente por gradiente. Valores comunes de α suelen ser 0,5, 0,9, 0,99.

Con la incorporación del término de momentum, ahora el paso de actualización depende también de la historia de los gradientes (más que nada la historia cercana), a lo largo del entrenamiento. De esta manera el paso dependerá de cuán grandes y cuán alineados fueron los gradientes pasados. En el caso de darse una alineación de los gradientes, por más que sus valores sean bajos, se producirá una aceleración de la actualización, acelerando la convergencia del aprendizaje. Este método ayuda también a evitar el estancamiento en mínimos locales.

Una variante del método de momentum, denominada **Nesterov Momentum**, fue introducida en [91] inspirada en el método del *gradiente acelerado de Nesterov*, [92, 93]. Donde la regla de actualización es:

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \gamma \nabla_W \Big(\frac{1}{M_B} \sum_i L_i(\boldsymbol{W} + \alpha \boldsymbol{v}) \Big), \tag{7.27}$$

$$\boldsymbol{W} \leftarrow \boldsymbol{W} + \boldsymbol{v} \tag{7.28}$$

Notar que este método realiza la actualización del gradiente de manera similar al método *momentum*, pero calculando el gradiente en el punto proyectado $\boldsymbol{W} + \alpha \boldsymbol{v}$ de acuerdo a la velocidad de la iteración anterior, o sea, se aplica primero la actualización por velocidad y luego se evalúa el gradiente. En el caso de BGD convexo, este método acelera la convergencia de $O(1/k)$ a $O(1/k^2)$, como se muestra en [92]; pero para el caso de SGD no mejora la velocidad de convergencia, [1].

A continuación se muestra un ejemplo de configuración de la *Train Chain* para estos métodos de optimización. Notar que se configuran a través de los sub-parámetros *momentum* (α) y *nesterov* (valor booleano):

```
{
"optimizer": "SGD",
"optimizer_params":{
        "learning_rate": 3e-3,
        "momentum": 0.5,
        "nesterov": true
        }
}
```

7.5.3. AdaGrad

El algorítmo **AdaGrad** (del acrónimo de *Adaptive Gradient*) pertenece al grupo de algoritmos adaptivos, donde su funcionamiento se basa en adaptar el learning rate de los parámetros del modelo mediante el escalamiento de los mismos de manera inversamente proporcional a la raíz cuadrática de la suma histórica de los valores del gradiente, según [94]. De esta manera, los parámetros que han tenido grandes gradientes respecto a la función de costo experimentarán una reducción fuerte de su learning rate, mientras que los parámetros que han tenido un gradiente pequeño tendrán una reducción mucho menor del learning rate. La regla de actualización para este algoritmo es:

$$\boldsymbol{g} \leftarrow \frac{1}{M_B}\nabla_W\Big(\sum_i L_i(\boldsymbol{W})\Big), \tag{7.29}$$

$$\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}, \tag{7.30}$$

$$\Delta\boldsymbol{W} \leftarrow -\frac{\gamma}{\epsilon + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}, \tag{7.31}$$

$$\boldsymbol{W} \leftarrow \boldsymbol{W} + \Delta\boldsymbol{W} \tag{7.32}$$

donde $\boldsymbol{r}$ es el acumulador del cuadrado de los gradientes $\boldsymbol{g}$, y ϵ una constante pequeña introducida por cuestiones de estabilidad numérica.

Si bien este método presenta buenas cualidades teóricas (al menos para optimizaciones convexas), en la práctica puede no funcionar muy bien para modelos de DL; donde usualmente finaliza prematuramente el entrenamiento, [1, 76]. Durante la evaluación de alguno de los modelos propuestos en este trabajo se utilizó este método con resultados no relevantes frente a los obtenidos con otros métodos.

A continuación se muestra un ejemplo de configuración de la *Train Chain* para este método de optimización.

```
{
"optimizer": "Adagrad",
"optimizer_params":{
        "learning_rate": 0.001,
        "initial_accumulator": 0.1,
        "epsilon": 1e-07
        }
}
```

7.5.4. RMSProp

El algoritmo **RMSProp** [95] modifica el algoritmo AdaGrad, para mejorar el desempeño en optimizaciones no-convexas, haciendo que la acumulación del gradiente ahora sea un promediado móvil pesado exponencialmente. AdaGrad converge rápidamente en funciones convexas, pero en redes neuronales rara vez nos encontraremos en esa situación; encontrando que AdaGrad suele estancarse en mínimos locales fácilmente y más aún cuando su learning rate es muy bajo debido a la acumulación monotónica que sufre. RMSProp, al usar ese promediado deslizante, va descartando los gradientes históricos, ajustándose mayormente a los valores más recientes, por lo que no sufre de esa disminución monotónica del learning rate y mantiene las cualidades de AdaGrad cuando se encuentra en superficies convexas de la función de costo. La regla de actualización para este algoritmo es:

$$\boldsymbol{g} \leftarrow \frac{1}{M_B}\nabla_W\Big(\sum_i L_i(\boldsymbol{W})\Big), \tag{7.33}$$

$$\boldsymbol{r} \leftarrow \rho\boldsymbol{r} + (1-\rho)\boldsymbol{g} \odot \boldsymbol{g}, \tag{7.34}$$

$$\Delta\boldsymbol{W} \leftarrow -\frac{\gamma}{\sqrt{\epsilon + \boldsymbol{r}}} \odot \boldsymbol{g}, \tag{7.35}$$

$$\boldsymbol{W} \leftarrow \boldsymbol{W} + \Delta\boldsymbol{W} \tag{7.36}$$

donde ρ es el parámetro que se introduce para realizar el promediado deslizante, denominándose *decay rate* (tasa de decaimiento), y ajusta la escala del promediado. En la práctica suele tomar valores como 0,9, 0,99 o 0,999.

Este algoritmo también puede incorporar el momentum en su actualización. En las siguientes ecuaciones se presenta la regla de actualización usando Nesterov Momentum (notar que sigue el mismo método presentado en 7.5.2):

$$\tilde{\boldsymbol{W}} \leftarrow \boldsymbol{W} + \alpha\boldsymbol{v}, \tag{7.37}$$

$$\boldsymbol{g} \leftarrow \frac{1}{M_B}\nabla_{\tilde{\boldsymbol{W}}}\Big(\sum_i L_i(\tilde{\boldsymbol{W}})\Big), \tag{7.38}$$

$$\boldsymbol{r} \leftarrow \rho\boldsymbol{r} + (1-\rho)\boldsymbol{g} \odot \boldsymbol{g}, \tag{7.39}$$

$$\boldsymbol{v} \leftarrow \alpha\boldsymbol{v} - \frac{\gamma}{\sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}, \tag{7.40}$$

$$\boldsymbol{W} \leftarrow \boldsymbol{W} + \boldsymbol{v} \tag{7.41}$$

A continuación se muestra un ejemplo de configuración de la *Train Chain* para este

método de optimización.

```
{
"optimizer": "RMSprop",
"optimizer_params":{
        "learning_rate": 0.001,
        "rho": 0.9,
        "momentum": 0.0,
        "epsilon": 1e-07
        }
}
```

7.5.5. Adam

El algoritmo **Adam** [96], es también adaptivo y su nombre deriva de *adaptive moments* (momentos adaptivos). Puede verse como una variación del algoritmo RMSProp. Primero, en Adam, el momento es incorporado directamente como una estimación del momento de primer orden (con pesado exponencial) del gradiente. Segundo, Adam incluye correcciones de bias a las estimaciones del momento de primer orden y del momento de segundo orden (el momento de segundo orden de RMSProp tiene un bias muy grande ni bien comenzado el entrenamiento) [1]. La regla de actualización para este algoritmo es:

$$\boldsymbol{g} \leftarrow \frac{1}{M_B}\nabla_W\Big(\sum_i L_i(\boldsymbol{W})\Big), \tag{7.42}$$

$$t \leftarrow t+1, \tag{7.43}$$

$$\boldsymbol{s} \leftarrow \beta_1\boldsymbol{s} + (1-\beta_1)\boldsymbol{g}, \tag{7.44}$$

$$\boldsymbol{r} \leftarrow \beta_2\boldsymbol{r} + (1-\beta_2)\boldsymbol{g}\odot\boldsymbol{g}, \tag{7.45}$$

$$\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1-\beta_1^t}, \tag{7.46}$$

$$\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1-\beta_2^t}, \tag{7.47}$$

$$\Delta\boldsymbol{W} \leftarrow -\gamma\frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}}+\epsilon}, \tag{7.48}$$

$$\boldsymbol{W} \leftarrow \boldsymbol{W} + \Delta\boldsymbol{W} \tag{7.49}$$

donde $\boldsymbol{s}$ es el momento de primer orden y $\boldsymbol{r}$ el de segundo orden, donde $\hat{\boldsymbol{s}}$ y $\hat{\boldsymbol{r}}$ son esos mismos momentos con la corrección de bias. Los valores recomendados son $\beta_1 = 0{,}9$, $\beta_2 = 0{,}999$ y $\epsilon = 10^{-8}$. Notar que la actualización es similar a la de RSMProp, pero usando una versión suavizada del momento (en este caso s).

Adam es un método muy robusto en general y, en general muestra mejores resultados que RMSProp, pero que en ocasiones hay que ajustar el learning rate respecto del sugerido, donde el learning rate sugerido es $\gamma = 0{,}001$. [97]

A continuación se muestra un ejemplo de configuración de la *Train Chain* para este método de optimización.

```
{
"optimizer": "RMSprop",
"optimizer_params":{
        "learning_rate": 0.001,
        "beta_1": 0.9,
        "beta_2": 0.999,
        "epsilon": 1e-07
        }
}
```

7.5.6. AdaDelta

El algoritmo **Adadelta** [98] es otra de las variantes del algoritmo Adagrad que busca evitar el problema de la reducción monotónica del learning rate por la acumulación de los cuadrados de los gradientes; y a su vez permite evitar la definición del hiper-parámetro learning rate global para el proceso de optimización. Este método recibe el nombre, probablemente, por *adaptive delta*; que tiene que ver con la novedad introducida por el algoritmo. AdaDelta considera, para la actualización de los pesos, un término dependiente de las actualizaciones realizadas sobre los pesos $\Delta \boldsymbol{W}$, afectadas por un promedio móvil.

La regla de actualización para este algoritmo es:

$$\boldsymbol{g} \leftarrow \frac{1}{M_B} \nabla_W \Big(\sum_i L_i(\boldsymbol{W}) \Big), \tag{7.50}$$

$$\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1-\rho)\boldsymbol{g} \odot \boldsymbol{g}, \tag{7.51}$$

$$\Delta \boldsymbol{W} \leftarrow \frac{\sqrt{\boldsymbol{w} + \epsilon}}{\sqrt{\boldsymbol{r} + \epsilon}} \odot \boldsymbol{g}, \tag{7.52}$$

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \gamma \Delta \boldsymbol{W}, \tag{7.53}$$

$$\boldsymbol{w} \leftarrow \rho \boldsymbol{w} + (1-\rho)\Delta \boldsymbol{W} \odot \Delta \boldsymbol{W} \tag{7.54}$$

donde $\boldsymbol{w}$ representa el cuadrado de las actualizaciones previas realizadas sobre los pesos, afectadas por el promedio móvil parametrizado por ρ al igual que los otros métodos. Notar que, respecto AdaGrad o RMSProp, al calcular $\Delta \boldsymbol{W}$ no se utiliza explícitamente

un learning rate (expresado como γ en otros métodos), sino $\sqrt{\boldsymbol{w} + \epsilon}$ que representa el valor RMS de $\boldsymbol{w}$; por lo que no sería necesario definir un learning rate global para este algoritmo. Sin embargo, en la práctica se deja disponible igualmente la parametrización del learning rate (inicial), nuevamente expresada aquí con el parámetro γ.

A continuación se muestra un ejemplo de configuración de la *Train Chain* para este método de optimización.

```
{
"optimizer": "Adadelta",
"optimizer_params":{
        "learning_rate": 0.001,
        "rho": 0.95,
        "epsilon": 1e-07
        }
}
```

En particular, para este trabajo, este optimizador ha sido el más frecuentemente utilizado; principalmente porque ha mostrado buenos resultados sin un ajuste fino de sus parámetros.

7.5.7. Otros optimizadores

Existen muchos otros optimizadores que no han sido presentados en esta sección, ya que se decidió poner el foco en los más relevantes y en los más intensivamente utilizados durante el desarrollo del proyecto. Se pueden listar algunos como: Adamax, Nadam, AMSGrad, FTRL, L-BFGS, SFO. Sin embargo la *Train Chain* soporta algunos de esos optimizadores como: Adamax, Nadam y FTRL.

Como la implementación de los modelos y la Train Chain se ha realizado con *Keras* [55] y *Tensorflow* [56], se puede recurrir a la información correspondiente para saber cuáles optimizadores son soportados y las parametrizaciones que soporta cada uno, visitando [99] y [100].

7.6. Métricas de evaluación

Las **métricas** son las unidades de medida que nos permiten evaluar el desempeño del modelo, durante el entrenamiento y una vez completado el mismo. La métrica que disponemos por defecto es el costo, presentado dentro del contexto del entrenamiento en 7.1 y luego detallado en relación a la función de costo que lo calcula, en 7.4. Usando la función *cross-entropy loss*, la función *Softmax* para la salida de nuestro modelo y

una codificación *one-hot* para las etiquetas, se llegó a la ecuación 7.20, a partir de la cual puede verse que el valor de costo está comprendido en el rango $(-\infty, 0]$, ya que su comportamiento está dominado por el logaritmo natural. Cuando una muestra se clasifica correctamente y con probabilidad 1, el costo es 0, por lo que el objetivo del entrenamiento será reducir el promedio del costo por cada muestra de la época hasta 0. Podría decirse que obtener un valor de costo promedio, por época, menor a 0,1 suele estar asociado a un desempeño aceptable; pero resulta que esta medida puede ser difícil de interpretar, y es por ello que resulta útil trabajar con algunas otras métricas aparte del costo.

Las métricas se calculan para todas las particiones del dataset: *train*, *validation* y *test*. Las métricas sobre el dataset *train* deberían siempre mejorar a lo largo de las épocas de entrenamiento, ya que siempre se busca optimizar la función de costo; salvo que algunos hiper-parámetros no hayan sido configurados adecuadamente, como por ejemplo, el *learning rate*. Monitorear el costo para el dataset de entrenamiento nos permite ajustar dichos hiper-parámetros para mejorar la velocidad de convergencia y evitar inestabilidades. Las métricas calculadas sobre los dataset de *validation* y *test* sirven exclusivamente para evaluar el desempeño del modelo para datos no utilizados para entrenarlo. Durante el entrenamiento es esperable que el costo para estas particiones también disminuya con las épocas, pero puede ocurrir que a partir de una determinada época el costo calculado sobre estas particiones comience a subir nuevamente; esto se debe a que el modelo empieza a sufrir de **overfitting** (sobre-ajuste), o sea, se ajusta cada vez mejor a los datos de entrenamiento pero no así para los datos desconocidos. En general, el entrenamiento puede cancelarse a partir de ese punto (detección de *overfitting* en las partición de *validation* o *test*), acción que se denomina **early-stopping** (detención temprana), puesto que no es útil mejorar el desempeño sobre la partición *train* a costa de las demás. La brecha entre el costo para la partición de entrenamiento y la de test es lo que se denomina **error de generalización** del modelo, y puede calcularse también con otras métricas.

Existe una diversidad grande de métricas que se utilizan en el campo de ML, pero las opciones se acotan de acuerdo al tipo de problema a resolver; por ejemplo, para problemas de regresión se suelen emplear métricas como: MSE, MAE, Root Mean Squared Error (Raíz del Error Cuadrático Medio) (RMSE); mientras que para los problemas de clasificación suelen emplearse algunas de las que se explicarán brevemente a continuación y que luego se utilizarán para mostrar los resultados obtenidos en este trabajo.

Accuracy (exactitud): Esta métrica indica cuál es la frecuencia en que las predicciones realizadas por el modelo coinciden con las etiquetas (clases) reales, o sea: $\boldsymbol{y}^{(i)} = \hat{\boldsymbol{y}}^{(i)}$.

Se calcula como la cantidad de coincidencias sobre el total de inferencias. Para nuestra aplicación, esta métrica es la que se prefiere como indicador de desempeño general del modelo. Como en nuestra aplicación hay más de una clase, codificadas como *one-hot*, esta métrica es equivalente a la métrica **Categorical Accuracy**.

Precision (Precisión): Esta métrica se calcula como la relación entre los verdaderos positivos (true positives) T_P y la suma de los verdaderos positivos y falsos positivos (false positives) F_P, o sea:

$$\text{precision} \triangleq \frac{T_P}{T_P + F_P} \tag{7.55}$$

Dicho de otra manera, esta métrica nos indica la proporción de veces que el modelo acertó en la inferencia de una clase respecto a todas las inferencias que el modelo informó como positivas para esa clase. Por ejemplo, una precisión del 85 % para la clase *animal*, nos dice que de todas las muestras en donde el modelo predijo *animal*, sólo el 85 % lo eran. Notar que en esta métrica no se tienen en cuenta aquellas muestras que eran *animal* (en el ejemplo) y no fueron clasificadas como tal (falsos negativos). Para más información ver [101, 102].

Recall (exhaustividad): Esta métrica se calcula como la relación entre los verdaderos positivos T_P y la suma de los verdaderos positivos y falsos negativos (false negatives) F_N, o sea:

$$\text{recall} \triangleq \frac{T_P}{T_P + F_N} \tag{7.56}$$

Dicho de otra manera, esta métrica nos indica la proporción de veces que el modelo acertó en la inferencia de una clase respecto a todas las inferencias que el modelo realizó sobre muestras que realmente pertenecían a esa clase. Por ejemplo, un recall del 90 % para la clase *car*, nos dice que de todas las muestras que eran un automóvil (car), el modelo predijo como *car* el 90 % de las veces. Notar que esta métrica no tiene en cuenta aquellas inferencias en que el modelo predijo que eran *car* y realmente no lo eran (falsos positivos). Para más información ver [101, 102].

F-score (valor-F): Es una métrica que combina las métricas precision y recall como la media harmónica de ambas, donde:

$$F_1 \triangleq \frac{2}{\text{precision}^{-1} + \text{recall}^{-1}} = 2\frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \tag{7.57}$$

Esta métrica entrega un valor cercano al promedio entre las dos tasas consideradas cuando ambos valores son similares, y entrega un valor más cercano al mínimo cuando ambos son disímiles. Suele ser una mejor métrica por clases en aspectos generales, que *precision* y *recall*, salvo que para la aplicación alguna de estas tenga especial interés (diganóstico de cancer, por ejemplo), o en casos en donde la cantidad de muestras por clases es muy desbalanceada dentro del dataset. Para más información ver [101–103].

Una vez entrenados los modelos, se considerarán los desempeños generales como los desempeño por clase. Para la comparación del desempeño por clase se hará uso, principalmente, de la **matriz de confusión**. Esta matriz muestra la proporción, o cantidad, de muestras de una clase particular que han sido inferidas como otra clase particular, mostrando todas las combinaciones posibles entre clases reales y clases inferidas. Esta herramienta es más útil para tener una idea si hay confusiones predominantes entre una clase real y una inferida.

7.7. Pos-procesamiento

Esta etapa no es una etapa que pertenezca literalmente al entrenamiento de los modelos, por ende a la *Train Chain*. En esta etapa se pos-procesan las inferencias realizadas por el modelo entrenado para entregar otro tipo de producto.

En lo que respecta a este trabajo, el pos-procesamiento que se realiza consiste en tomar las inferencias realizadas para muestras (frames) consecutivas en el tiempo, que pertenecen a una adquisición radar realizada sobre un blanco en particular; y generar una inferencia a lo largo del tiempo de adquisición utilizando la evolución histórica de las inferencias realizadas por cada muestra. Este proceso permite filtrar malas inferencias realizadas por frames de mala calidad presentes durante la adquisición, y mejorar el desempeño general del clasificador partiendo del hecho de que por cada adquisición se supone que hay un sólo tipo de blanco iluminado por el radar sobre el cual se hizo la extracción de la señal Doppler.

Durante el pos-procesamiento también se puede evaluar la calidad del frame a clasificar, descartando las inferencias realizadas para estos frames. El descarte se realiza tal como se implementó en la Dataset Chain para descartar segmentos donde la calidad de la señal Doppler no era buena. Esto introduce una mejora adicional en el producto que se entrega al usuario, incrementando la confiabilidad de los resultados.

Los resultados obtenidos usando pos-procesamiento y que se presentarán en la sección siguiente, muestran un mejor desempeño del clasificador, ya sea por clase como de manera general; por lo que las métricas más conservadoras serán aquellas mencio-

nadas en la sección anterior, y serán las que se usarán para validar los requerimientos de desempeño. Sin embargo, es valioso analizar estos valores entregados por el pos-procesador, ya que serán potencialmente los que se mostrarán al usuario durante la operación del radar.

Capítulo 8

Resultados

A lo largo del documento se ha explicado la naturaleza de los datos, los tipos de pre-procesamiento de las señales, el diseño de la cadena para conformar el dataset, las arquitecturas de los modelos de clasificación y sus hiper-parámetros; la cadena de entrenamiento y las métricas de desempeño. Eso nos permite proseguir con el entrenamiento de los modelos (con diversas hiper-parametrizaciones) utilizando los distintos datasets conformados. Tener en cuenta, como ya fue explicado anteriormente, los resultados más importantes serán los que se obtengan para la partición *test* de cada dataset.

En este capítulo se mostrarán los resultados más relevantes de todas las sesiones definidas a lo largo del desarrollo del trabajo. Vale mencionar que las sesiones que se consideran relevantes son aquellas en donde se obtuvieron los mejores resultados; pero también se muestran los resultados de sesiones que permiten ver la variabilidad de las métricas en función de parametrizaciones o modelos particulares. En general, mucha de la diversidad en configuraciones, se mostrará utilizando un mismo modelo (el que mostró mejor desempeño entre los evaluados), para evitar que la combinatoria de las opciones dificulte la visualización de los resultados y la obtención de conclusiones.

Muchos datasets, modelos e hiper-parametrizaciones han sido evaluadas a lo largo del trabajo, pero la estrategia de desarrollo no ha sido generar un Montecarlo de todas (o la mayoría) de las variables que intervienen a lo largo de todas las cadenas, desde el dato crudo hasta la entrega del resultado de la inferencia; sino que se fue evaluando la sensibilidad a algunos de los parámetros y arquitecturas, buscando la optimización de las métricas de interés, para así ir estableciéndolas como definitivas y poder continuar evaluando algunas otras combinatorias de otras variables. Esto mismo revela que hay muchas posibles arquitecturas y parametrizaciones que no fueron evaluadas y que podrían brindar mejores resultados incluso; pero sería necesario abordarlas con una estrategia de exploración (Montecarlo o similar), lo que demandaría de una infraestructura de cómputo muy grande y de ampliar la funcionalidad de las cadenas

implementadas para poder realizarlo.

A partir de los resultados obtenidos se presentarán algunas conclusiones parciales que permiten justificar algunas de las líneas exploradas; pero algunas otras conclusiones muestran que existen muchas líneas más que resultan interesantes de explorar con mayor profundidad. Sin embargo, las conclusiones generales y las líneas para trabajos futuros se presentarán recién en la parte III.

8.1. Comparativa entre modelos

En la sección 6.9 se presentaron los modelos considerados como relevantes, donde el modelo **grillo** es una red neuronal convencional de dos capas ocultas, el modelo **granadero** es una red del tipo CNN chica, y el modelo **pollito** es una red también del tipo CNN pero de tamaño moderado. Los tres modelos fueron entrenados con el dataset *corona*, usando el mapeo #4, cuyas características están listadas en la tabla 5.1. Estos resultados se muestran en la figura 8.1. Los modelos fueron entrenados utilizando un optimizador *Adadelta* con $\gamma = 0{,}1$ para pollito, $\gamma = 0{,}01$ para granadero y $\gamma = 0{,}001$ para grillo, con un mini-batch de 64 muestras para pollito y granadero y 128 muestras para grillo. Estos parámetros fueron elegidos en función de mejorar la convergencia y estabilidad a lo largo del entrenamiento.

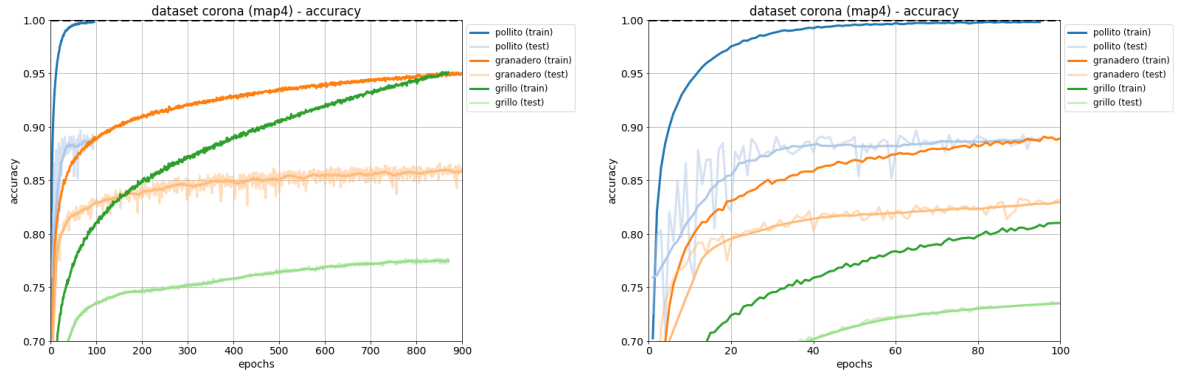


Figura 8.1: Comparación entre modelos usando el dataset *corona map4*. El gráfico muestra las curvas de accuracy vs epochs, para el train dataset y test dataset. En la figura (b) se muestran las primeras 100 épocas. Las curvas de test se muestran también con un suavizado usando una ventana deslizante rectangular de 20 muestras.

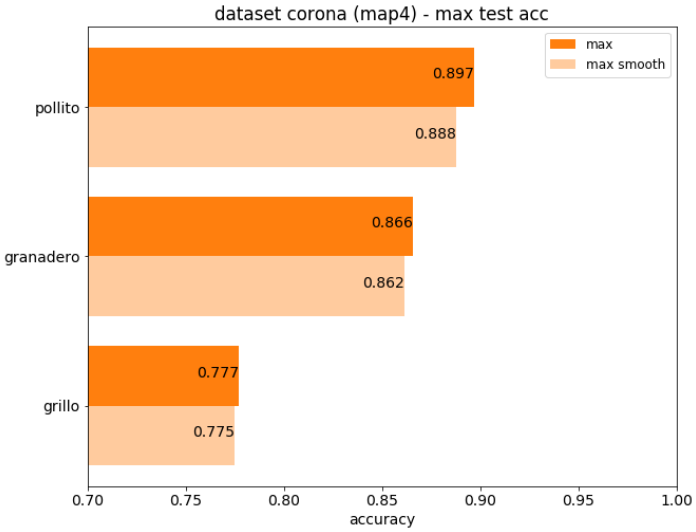


Figura 8.2: Valor máximo de accuracy (test) obtenido entre modelos usando el dataset *corona map4*.

De los resultados obtenidos puede observarse que el modelo *pollito* es el que mejor desempeño obtuvo para la partición de test, el modelo *granadero* obtuvo resultados satisfactorios para los requerimientos de la aplicación, mientras que el modelo *grillo* no alcanzó a cumplir dichos requerimientos (superar el 80 % en la métrica *accuracy* para la partición test). El máximo valor obtenido para cada caso se muestra en la figura 8.2.

Vale mencionar que el entrenamiento de los distintos modelos se aborta luego de una determinada cantidad de épocas sin mejoras en las métricas sobre la partición de test, lo que se denomina *early-stopping with patience*. Se ha experimentado prolongar el entrenamiento de los modelos *granadero* y *grillo*, consiguiendo alcanzar aproximadamente el 100 % de accuracy para la partición train, pero sin mejoras en las métricas sobre la partición test. A diferencia de estos, el modelo *pollito*, de mayor capacidad, puede converger en el entrenamiento en muchas menos épocas, alrededor de 60 usando el optimizador *Adadelta* con $\gamma = 0{,}1$. Se podría concluir que la capacidad del modelo ha permitido mejorar el desempeño y el tiempo de convergencia, pero a costa de sufrir overfitting (sobre-ajuste). Este sobre ajuste puede verse en la figura 8.3, donde se muestra el valor de loss por cada época de entrenamiento; donde el punto de overfitting se da cuando el valor de loss para la partición de test comienza a crecer nuevamente (esto se da alrededor de la época 20). A partir de ese punto se logra una mejora leve de desempeño en *accuracy* para luego establecerse en el valor de convergencia.

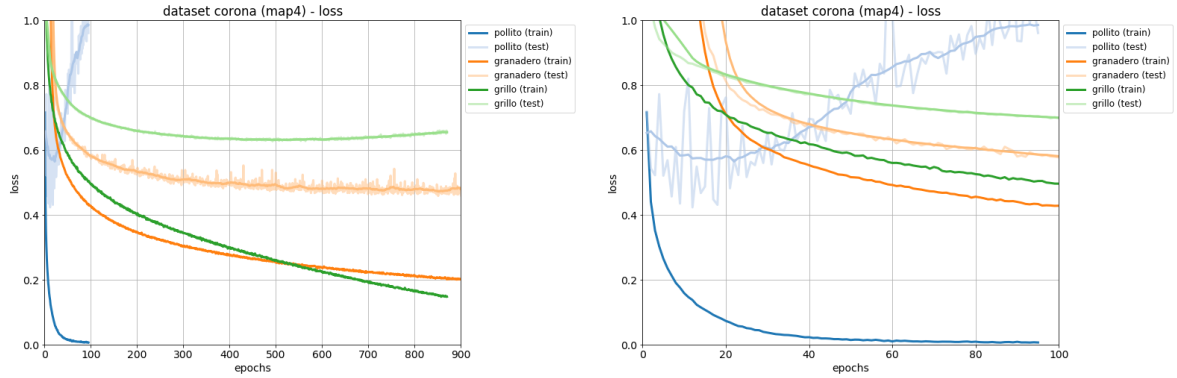


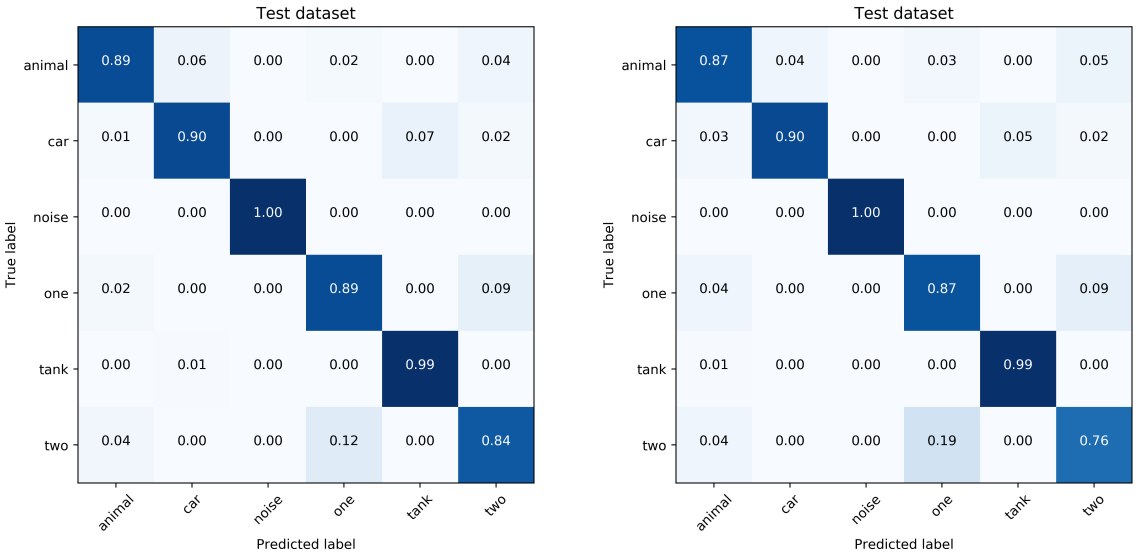
Figura 8.3: Comparación entre modelos usando el dataset *corona map4*. El gráfico muestra las curvas de loss vs epochs, para la partición train y test del dataset. En la figura (b) se muestran las primeras 100 épocas. Las curvas de test se muestran también con un suavizado usando una ventana deslizante rectangular de 20 muestras.

El marcado overfitting que sufre el modelo *pollito* nos indica que la capacidad del modelo es demasiado grande para la aplicación y es por ello que el modelo *granadero* es una buena alternativa si la plataforma en donde se implementará no dispone de una capacidad de cómputo grande. Para el caso del modelo *grillo*, hay overfitting igualmente, que se da alrededor de la época 500, lo cuál indica que es igualmente grande para la aplicación.

Los modelos *pollito* y *grillo* tienen una cantidad de pesos similares, pero hay una gran diferencia en el desempeño entre ambos modelos. Más allá que el modelo *grillo* no ha podido alcanzar el 80 % de accuracy (test), el error de generalización es igualmente grande; o sea, el modelo está ajustándose bien para los datos de entrenamiento sin lograr lo mismo para los datos de test. Esto último justifica el uso de redes del tipo CNN frente a las redes neuronales convencionales. El modelo *granadero*, con muchos menos parámetros ha superado en desempeño ampliamente al modelo *grillo*, como puede verse en la figura 8.2.

8.1.1. Métricas por clase

Si nos enfocamos en los dos modelos implementados con CNN, para evaluar el desempeño por clase se puede utilizar matrices de confusión. En la figura 8.4 se representan las matrices de confusión para la partición test. Notar que para ambos modelos las clases *noise* y *tank* no presentan problemas en la clasificación, mientras que la clase que presenta la mayor dificultad es *two*. El resto de las clases presentan un desempeño muy similar entre ambos modelos.



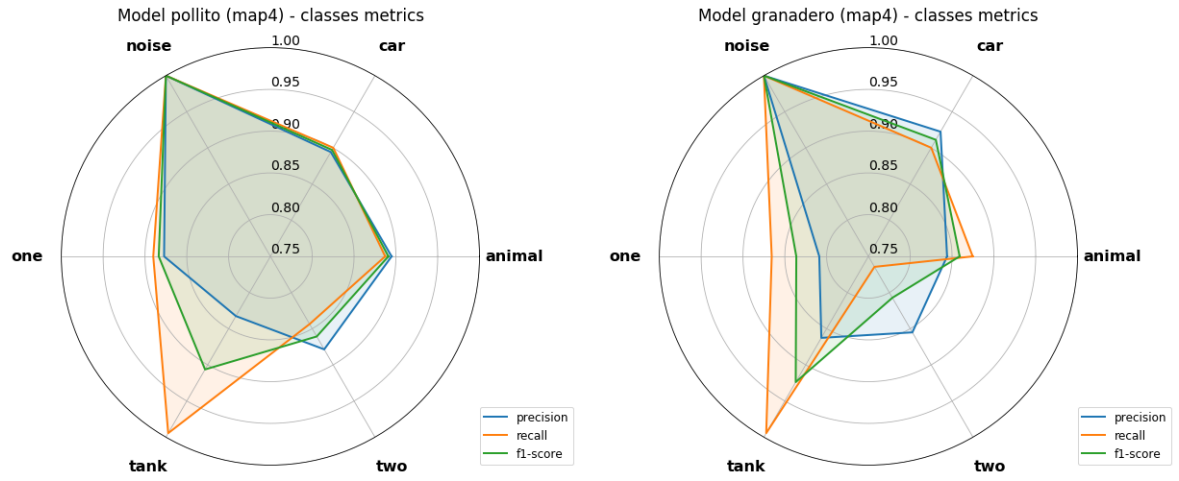
(a) pollito **(b)** granadero

Figura 8.4: Matrices de confusión correspondientes a la partición *test*, para los modelos *pollito* (a) y *granadero* (b).

Analizando los valores de las matrices de confusión, se puede observar que las confusiones más probables suelen darse entre las clase *one* y *two*, y en un porcentaje menor con *animal*. Intuitivamente esto tiene cierta lógica, ya que muchas adquisiciones tienen dos personas caminando sincrónicamente; y un animal cuadrúpedo puede presentar componentes Doppler similares a la de dos personas. Tener en cuenta, que para una

aplicación real de vigilancia, no tiene mucha utilidad discriminar entre las clases *one* y *two*, por lo que utilizando otro mapeo donde ambas clases se conjugan en *person*, el clasificador tendrá mejor performance. Sí es más importante discriminar entre *person* (*one* o *two*) y *animal* para evitar falsas alarmas. En el caso del modelo *pollito* se nota una mejora significativa en el desempeño sobre la clase *two*, que es la que termina marcando principalmente la diferencia en el desempeño general entre los modelos.

En la sección 7.6 se presentaron las métricas *precision*, *recall* y *F-score* como las métricas que se usarán para evaluar el desempeño de los modelo por cada clase. En la figura 8.5 se grafican en forma polar estas métricas para los modelos *pollito* y *granadero*. Notar que la clase *tank* presenta, en ambos casos, una diferencia importante entre precision y recall, donde (según las matrices de confusión) la principal clase de confusión es *car*. *Granadero* presenta igual diferencia para la clase *two*, en donde ya se mencionó que su desempeño no era del todo bueno, y en menor medida para la clase *one*. Estas diferencias, para las clases one y two no son marcadas en el caso del modelo pollito, por lo que podemos concluir que la mejora de desempeño entre ambos modelos se da gracias a una mejor clasificación de las clases *one* y *two*.



(a) pollito **(b)** granadero

Figura 8.5: Métricas precision, recall y F1-score sobre la particion *test*, para los modelos *pollito* (a) y *granadero* (b) para el dataset *corona map4*.

Si nos concentramos en la métrica F1-score, que representa un promedio de las métricas precision y recall, podemos comparar nuevamente ambos modelos y observar que las clases *one* y *two* marcan las principales diferencias en desempeño, tal como puede observarse en la figura 8.6. En la misma figura se incluyó el desempeño del modelo *grillo* para poder visualizar la gran diferencia de desempeño entre este y los modelos CNN. Las diferencias más grandes vuelven a darse para las clases *one* y *two*, mientras que para las clases *animal*, *car* y *tank* presenta un desempeño aceptable. Notar que la clasificación de las muestras con sólo ruido ha sido excelente, al igual que

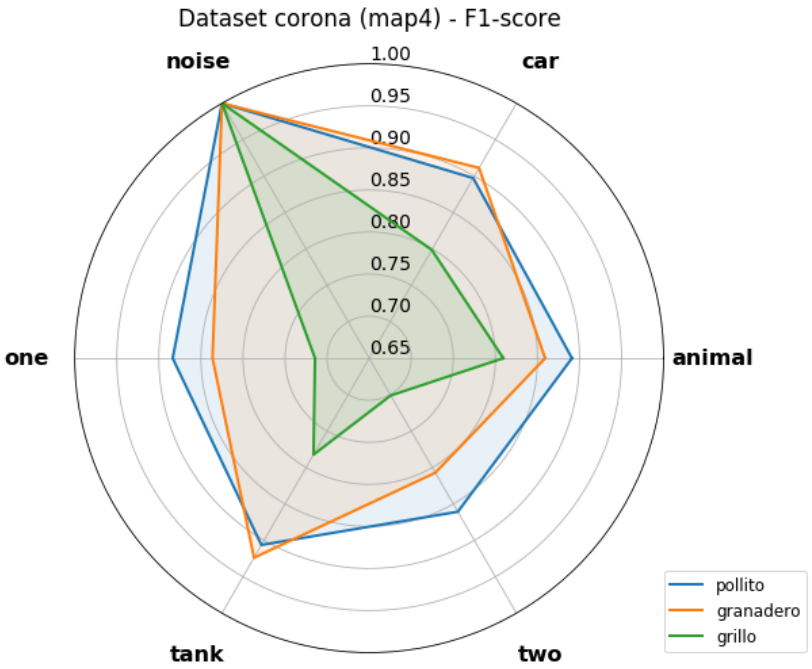


Figura 8.6: Comparación del desempeño por clase entre los modelos *pollito*, *granadero* y *grillo*, utilizando la métrica F1-score.

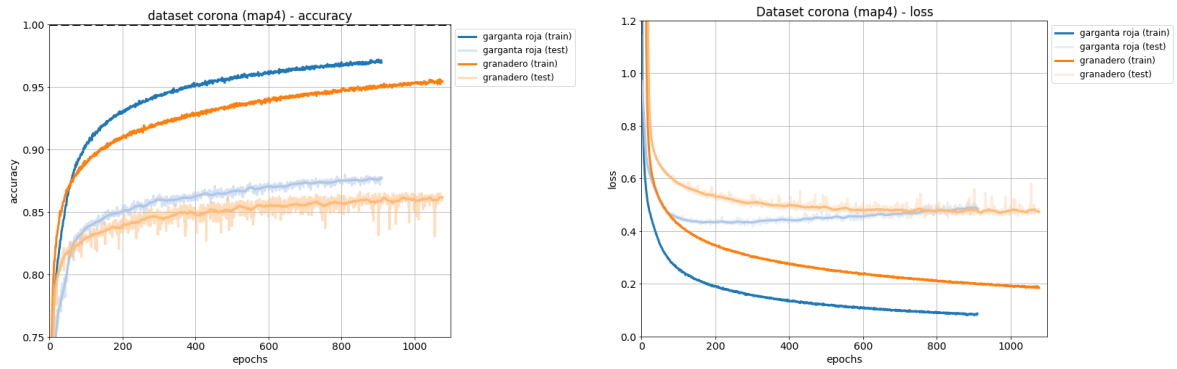
los otros modelos.

8.1.2. Regularizaciones de modelos

En el proceso de experimentación con distintas arquitecturas, a los modelos no se les incorporó regularización interna, como las descriptas en 7.4.1. Una vez que una arquitectura presentaba un buen desempeño (excelente desempeño para la partición *train* y aceptable para la partición *test*), se incorporaron algunas regularizaciones internas para intentar reducir el error de generalización.

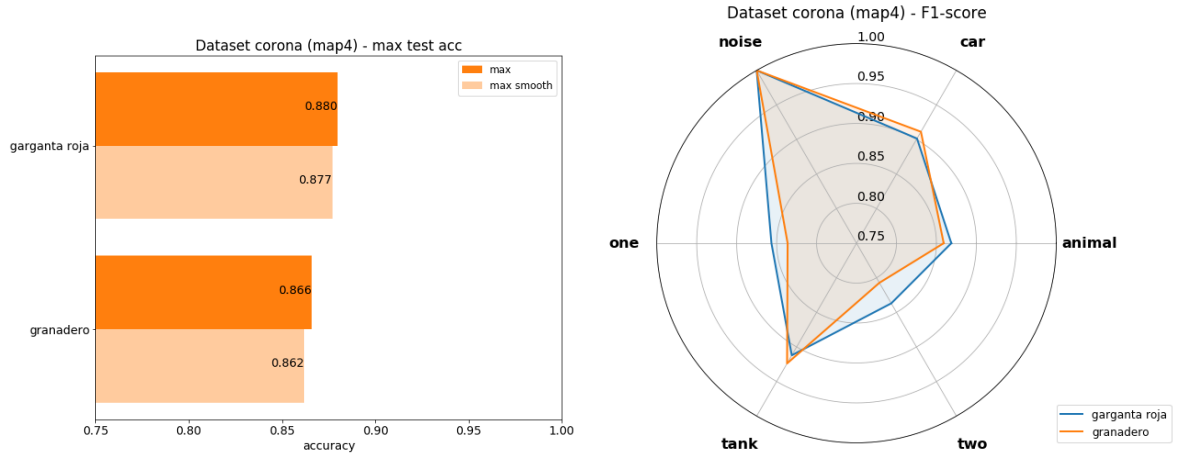
El modelo *granadero* surge de la incorporación de regularización interna del modelo *garganta roja*, en donde se incorporó *Batch Normalization* en las capas convolucionales y *kernel regularization* del tipo L^2 para la etapa de clasificación (capas densas). De la misma manera, se evaluaron modelos derivados de *pollito*, como ser *gallito* en donde se incorporó igualmente *Batch Normalization* y *kernel regularization*. En ninguno de los casos la incorporación de regularización dentro del modelo presentó mejoras sobre el desempeño de la partición *test*. Es importante mencionar que todos los modelos aquí mencionados (con o sin regularización) hacen uso de la regularización *Dropout*.

La introducción de este tipo de regularizaciones sí redujo el error de generalización durante el entrenamiento, ya que durante las épocas tempranas del entrenamiento, la diferencia en accuracy entre train y test era menor; pero luego el desempeño para test convergía alrededor de los mismos valores que lo hacía el modelo sin regularización,



(a) accuracy **(b)** loss

Figura 8.7: Comparación de los modelos *garganta roja* (sin regularización) y *granadero* (con regularización). Ambos casos fueron entrenados usando un optimizador Adadelta con $\gamma = 0{,}1$.

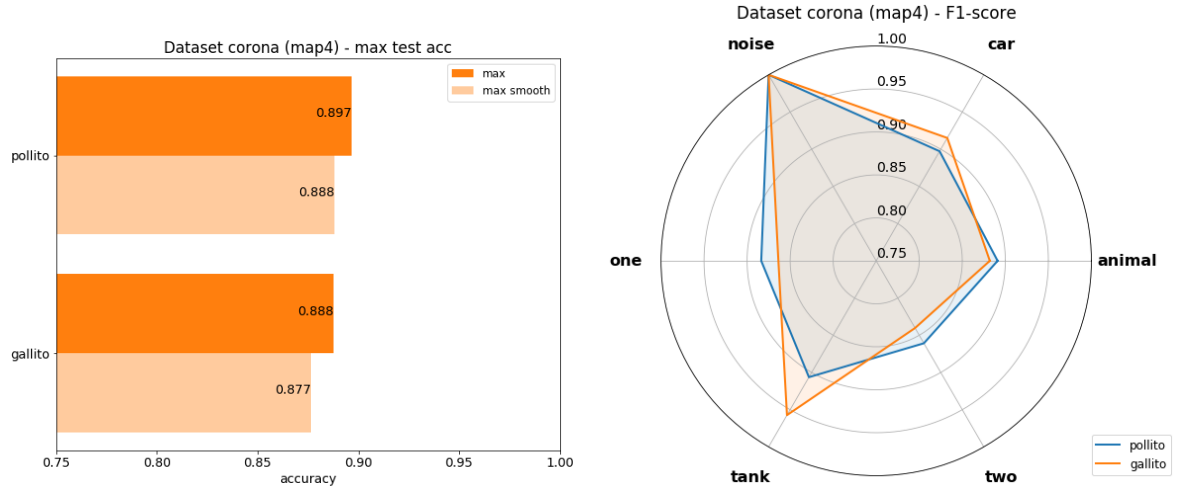


(a) accuracy **(b)** loss

Figura 8.8: Comparación de los modelos *garganta roja* (sin regularización) y *granadero* (con regularización).

y el entrenamiento restante volvía a incrementar (en menor medida) el error de regularización. Esto puede observarse en la figura 8.7. La incorporación de este tipo de regularizaciones también generaba más inestabilidad en los valores de desempeño de test (tanto *accuracy* como *loss*), lo que dificultó el ajuste de los parámetros de los optimizadores, que en general se soluciona disminuyendo el learning-rate, ralentizando el entrenamiento. En la misma figura, en las curvas de *loss*, se puede notar como se previene el overfitting en el caso donde se introduce la regularización.

En las figuras 8.8 y 8.9 se muestran los resultados de test *accuracy* y *F1-score* por clase, para los casos de los modelos *garganta roja* y *pollito* correspondientemente, comparativamente con sus versiones regularizadas. Es interesante notar, que a pesar de ser arquitecturas diferentes, la introducción de la regularización genera un patrón muy similar en los resultados. Uno de ellos es la reducción de la *test accuracy* entre el 1 % y 2 % (relativo) al introducir regularización. El otro, es degradar el desempeño para las clases *one*, *two* y *animal*, para mejorarlo en las clases *tank* y *car*.



(a) test accuracy **(b)** F1-score

Figura 8.9: Comparación de los modelos *pollito* (sin regularización), y *gallito* (con regularización). En (a) se muestran los mejores valores de desempeño para test, y en (b) se muestra el F1-score por clase.

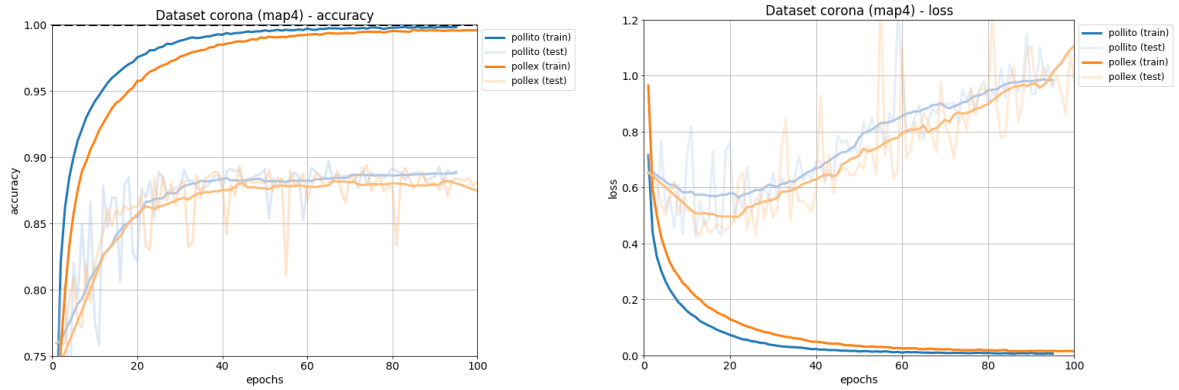
Una primera conclusión sobre la regularización es que efectivamente está reduciendo el error de generalización al inicio del entrenamiento, pero que afecta a la convergencia del entrenamiento. Quizás se deba explorar aún más sobre el ajuste del optimizador y la función de costo para lograr realizar un entrenamiento con una velocidad de convergencia más elevada y que no haya una penalización sobre algunas clases en particular. En esa línea, se podría evaluar de ponderar de manera desequilibrada el aporte de cada clase en la función de costo.

8.1.3. Capacidad del modelo

Una de las técnicas para reducir el overfitting a costa del desempeño de la partición *train*, es reducir la capacidad del modelo; similar a lo que realiza la técnica de regularización *Dropout*. Se analizó entonces reducir la capacidad del modelo de mejor desempeño, pero elevado overfitting, intentando que esto mejore la capacidad de generalización del mismo.

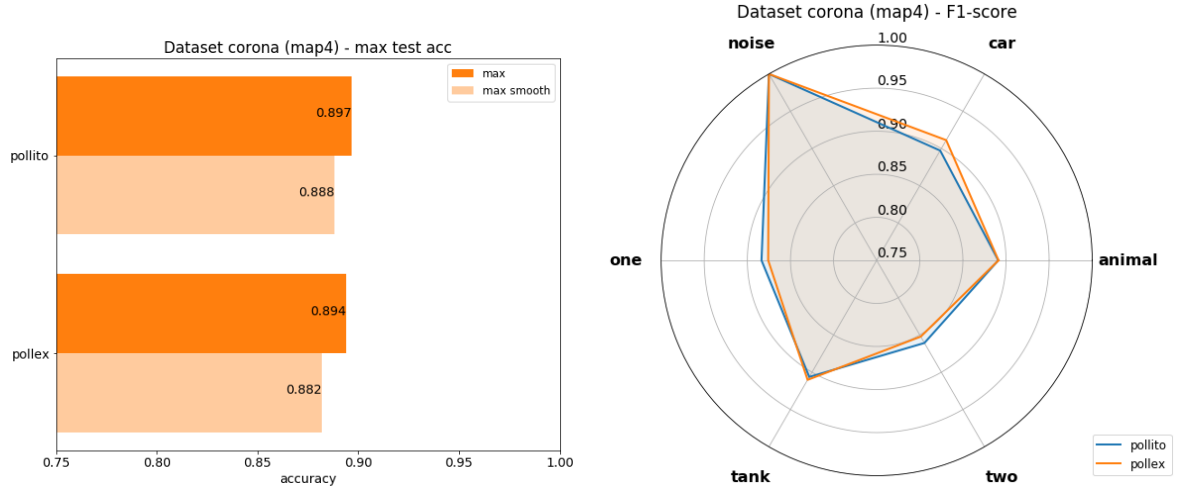
Como el modelo *granadero* ya es un modelo con menor capacidad entre los evaluados, donde hay menos capacidad tanto en la etapa de *features learning* como la de *classification*; se decidió crear una variante del modelo *pollito*, denomina **pollex**, en donde se reduzca fuertemente la capacidad de la etapa de *classification*, o sea, el tamaño de las redes densas. Con esa reducción, el modelo *pollex* quedó con casi la misma cantidad de pesos que el modelo *granadero*. Para más detalles ver 6.9.

Los resultados arrojados en este análisis son interesantes, puesto que el desempeño obtenido con el modelo reducido es muy similar al obtenido con el modelo *pollito*, por



(a) accuracy **(b)** loss

Figura 8.10: Comparación de los modelos *pollito* (mayor capacidad, 3 805 702 pesos en la etapa de classification), y *pollex* (con menor capacidad, 427 014 en la misma etapa).



(a) test accuracy **(b)** F1-score

Figura 8.11: Comparación de los modelos *pollito* (mayor capacidad, 3 805 702 pesos en la etapa de classification), y *pollex* (con menor capacidad, 427 014 en la misma etapa). En (a) se muestran los mejores valores de desempeño para test, y en (b) se muestra el F1-score por clase.

lo que también supera en casi igual medida al modelo *granadero* (con una cantidad de parámetros similar). Notar los valores de desempeño general y por clase, presentados en la figura 8.10; donde las diferencias de desempeño del modelo *pollex* están muy levemente por debajo de las de *pollito*. Durante el entrenamiento, en igualdad de condiciones, el modelo *pollex* tiene un tiempo de convergencia levemente menor, pero igualmente alcanza valores casi óptimos para la partición *train* del dataset.

En base a estos resultados se puede concluir que si es necesario reducir el tamaño del modelo por cuestiones vinculadas a la implementación en la plataforma radar, el modelo *pollex* es la alternativa recomendada entre los modelos evaluados. Por otro lado, también se puede ver que la capacidad de la etapa de *classification* ha marcado muy poca diferencia en el desempeño, no siendo así el caso de la etapa de *features learning*; donde su arquitectura y capacidad han sido más determinantes.

8.1.4. Tiempo de inferencia

Para tener un orden de magnitud del tiempo de pre-procesamiento y de inferencia del clasificador propiamente dicho, en la tabla 8.1 se presentan los resultados obtenidos en una PC con un procesador Intel i7 8va generación (i7-8750H CPU @ 2.20GHz × 12), 16 GiB de memoria RAM, GPGPU Nvidia GeForce GTX 1060 (6078 MiB), Nvidia Driver 440.100, versión de CUDA 10.2; con un sistema operativo Ubuntu 18.04.5 LTS de 64 bits.

	mean time *[ms]*	*std time* *[ms]*	*throughput* *[sps]*	*mean time* *[PRIs]*
Pre-Processing	1.95	0.17	512	11.1
Inference	20.45	7.98	48.9	116.2
Total	22.41	7.98	44.6	127.3

Tabla 8.1: Desempeño en tiempo de pre-procesamiento (espectrogram *fixdim*), tiempo de inferencia y tiempo total de generación del reporte de clasificación para una muestra; utilizando el modelo pollito. La columna de throughput indica la cantidad de muestras que se procesan por segundo. El tiempo medio expresado en PRI expresa la cantidad de pulsos (intervalos) que el radar adquiere durante el tiempo que demandó procesar una muestra en la etapa correspondiente (considerando la PRF de 5681.8 Hz).

Si bien los resultados mostrados en la tabla 8.1 se corresponden al modelo *pollito*, para el resto de los modelos, los tiempos obtenidos son similares. Esto se debe a que la mayor parte del tiempo se consume en el movimiento de datos desde la memoria RAM de la CPU a la GPU, por lo que el tiempo de inferencia propiamente dicho queda enmascarado por este último. También puede observarse en la tabla que el tiempo de conformación de la imagen, usando el pre-procesamiento espectrograma *fixdim*, es casi el 10 % del tiempo de inferencia. Tener en cuenta que el pre-procesamiento se realiza en la CPU. El tiempo total se podría reducir aún más realizando una implementación ajustada a la plataforma de procesamiento radar que se disponga y trasladando la etapa de pre-procesamiento a la GPGPU.

El tiempo total de pre-procesamiento e inferencia es menor a los 100 ms requeridos en los objetivos (ver 1.2) sin una implementación optimizada. A su vez, con este tiempo total es posible generar reportes de clasificación a una tasa mayor a 44 muestras (frames) por segundo. Es útil mirar el valor del tiempo total medio en función de la cantidad de pulsos generados por el radar (de acuerdo a su tiempo entre pulsos o PRI), ya que nos indica la cantidad de muestras nuevas que se generarían mientras la cadena de clasificación está realizando la inferencia. Este valor nos indicaría el salto mínimo

(*stride*: $N_H = W_{len} - N_{ov}$, ver 5.1.1) en número de muestras de la señal Doppler que permitirían realizar inferencias continuas si se está clasificando en tiempo real.

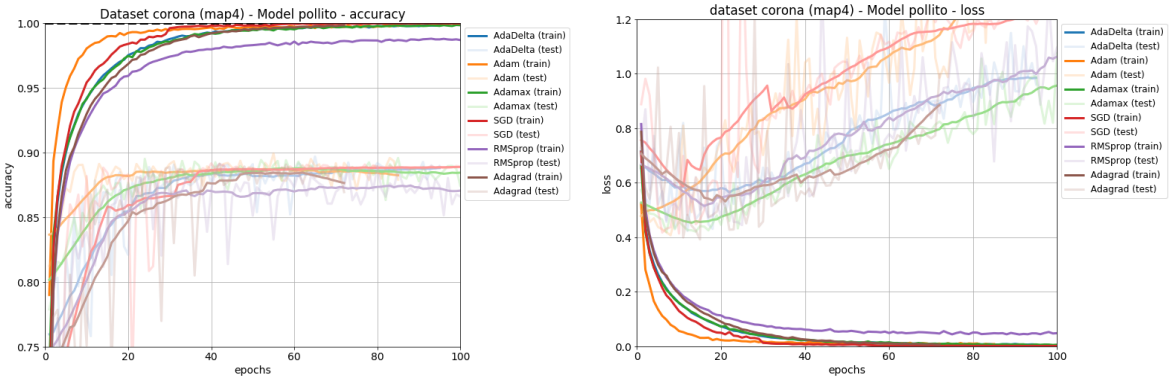
8.2. Comparativa entre optimizadores

Una de las etapas exploratorias del trabajo fue evaluar diversos optimizadores y parametrizaciones de los mismos. Algunos de los optimizadores evaluados fueron presentados en mayor detalle en la sección 7.5. La selección de optimizadores fue realizado teniendo en cuenta los que son más utilizados en aplicaciones de clasificación de imágenes, utilizando redes del tipo CNN. Los optimizadores que se muestran en esta comparación son: *AdaDelta*, *Adam*, *Adamax*, *SGD*, *RMSprop* y *AdaGrad*.

El proceso de evaluación de estos optimizadores fue compararlos utilizando un mismo dataset, partir de la parametrización más común o la recomendada en el paper correspondiente, y luego ajustar dichos parámetros para mejorar la velocidad de convergencia sin comprometer la estabilidad de la optimización ni el valor final de desempeño (valor de convergencia). El dataset elegido para mostrar los resultados comparativos es *corona*, nuevamente usando el mapeo #4.

Las parametrizaciones que se usaron en cada uno de los casos son:

- **AdaDelta**, ver 7.5.6:
  - $\gamma = 0{,}1$
  - $\rho = 0{,}95$
  - $\epsilon = 10^{-7}$
- **Adam**, ver 7.5.5:
  - $\gamma = 3 \cdot 10^{-4}$
  - $\beta_1 = 0{,}9$
  - $\beta_2 = 0{,}999$
  - $\epsilon = 10^{-7}$
- **Adamax**:
  - $\gamma = 3 \cdot 10^{-4}$
  - $\beta_1 = 0{,}9$
  - $\beta_2 = 0{,}999$
  - $\epsilon = 10^{-7}$
- **SGD**, ver 7.5.1 y 7.5.2:
  - $\gamma = 0{,}05$ (se reduce $\times 0{,}5$ al estancarse test accuracy)
  - $\alpha = 0{,}0$ (momentum)
  - Nesterov = False
- **RMSprop**, ver 7.5.4:
  - $\gamma = 5 \cdot 10^{-5}$
  - $\rho = 0{,}9$
  - $\alpha = 0{,}0$ (momentum)
  - $\epsilon = 10^{-7}$
- **AdaGrad**, ver 7.5.3:
  - $\gamma = 0{,}01$
  - $r = 0{,}1$ (valor inicial)
  - $\epsilon = 10^{-7}$



(a) accuracy **(b)** loss

Figura 8.12: Comparación de optimizadores utilizando el modelo *pollito* y el dataset *corona map4* a lo largo de las épocas de entrenamiento.

De los resultados obtenidos para cada una de las sesiones de entrenamiento puede concluirse que los optimizadores tiene un desempeño muy similar, con algunas diferencias en el comportamiento a lo largo del entrenamiento. El caso del optimizador *RMSprop* fue el que menor desempeño mostró, pero sin alejarse mucho del resto.

AdaDelta presenta un buen desempeño general y valores balanceados para las métricas por clase, con el mejor desempeño para la clase *two*. Este optimizador fue utilizado por defecto en muchas de las etapas de experimentación debido a su fácil parametrización. Por otro lado, **Adamax** presenta un desempeño general casi idéntico, pero con una mejor estabilidad y menor valor de la función de costo. En cuanto a las métricas por clase, presenta una gran mejora para las clases *tank* y *car*, respecto a *Adadelta*; pero levemente peor en *one* y *two*. **AdaGrad**, con una curva similar para la partición *train*, muestra un desempeño medio levemente menor durante las primeras etapas del entrenamiento, pero en los valores finales no muestra una diferencia significativa.

Adam presenta el menor tiempo de convergencia, siendo este tiempo muy insensible a la parametrización, pero presentando el mayor overfitting; como puede verse en 8.12. En cuanto al desempeño por clase, supera a los demás clasificadores en todas las clases con excepción de la clase *two*. Este optimizador parece recomendable para evaluar nuevos modelos, ya que presenta un buen desempeño general y una buena insensibilización a su parametrización.

Respeto al optimizador **RMSprop**, su desempeño fue menor que los mencionados anteriormente, con la particularidad de que no se logró converger para la partición *train* en pocas épocas de entrenamiento. Notar que en los resultados mostrados en la figura 8.13 los valores de desempeño de test están muy cercanos a lo obtenido en los optimizadores anteriores. Su parametrización resultó más dificultosa para lograr una velocidad de convergencia aceptable sin comprometer demasiado la estabilidad de las

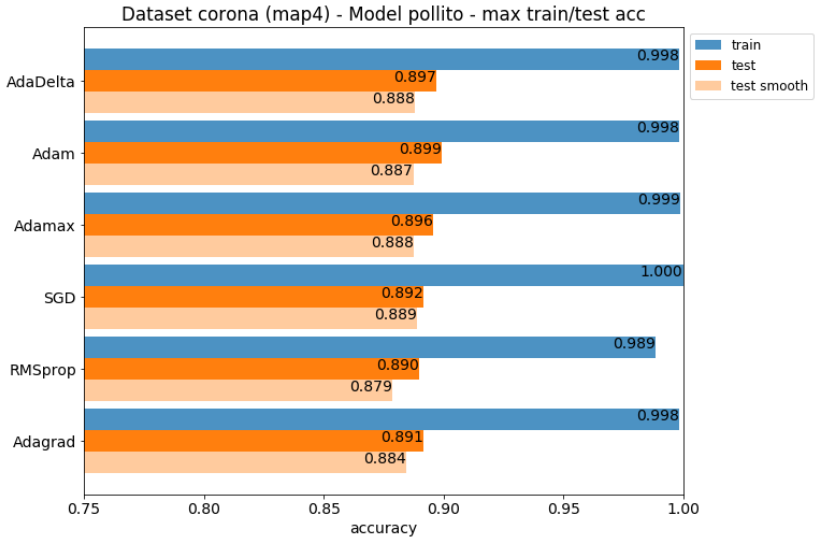


Figura 8.13: Comparación de los mejores valores de desempeño obtenidos para cada uno de los optimizadores evaluados con el dataset *corona map4*, tanto para la partición *train*, como *test*.

métricas sobre la partición *test*.

El optimizador **SGD** requirió realizar una variación del learning-rate a lo largo de las épocas de entrenamiento para mejorar el tiempo de convergencia. Estas modificaciones del learning-rate pueden verse como pequeños escalones en las curvas mostradas en las gráficas de la figura 8.12. La sintonización del learning-rate fue más trabajosa para poder asemejarse a la de los optimizadores adaptivos. Al no realizarse esta sintonización, la convergencia demandaba muchas épocas de entrenamiento.

En función de todos los experimentos realizados y los resultados obtenidos, la elección del optimizador no es un factor determinante en la consecución de un buen desempeño (para este dataset y modelos evaluados). Por eso se optó por utilizar un optimizador adaptivo, como *AdaDelta* por ejemplo, para realizar la mayoría de los demás experimentos y ajustes; pero es un tema a reconsiderar una vez que se quiera conseguir aún mejores resultados.

8.3. Comparativa entre datasets

Se han evaluado diversas parametrizaciones y técnicas de pre-procesamiento para analizar la sensibilidad del desempeño a dichas variables. Así se ha conformado un conjunto de datasets, listados y detallados en 5.7, que permitieran entrenar los distintos modelos propuestos (y futuros que puedan proponerse) para sacar algunas conclusiones al respecto. En la tabla 5.7 se resaltan las diferencias respecto del dataset **corona**, que se toma como referencia.

Los resultados aquí presentados se corresponden al entrenamiento de los datasets

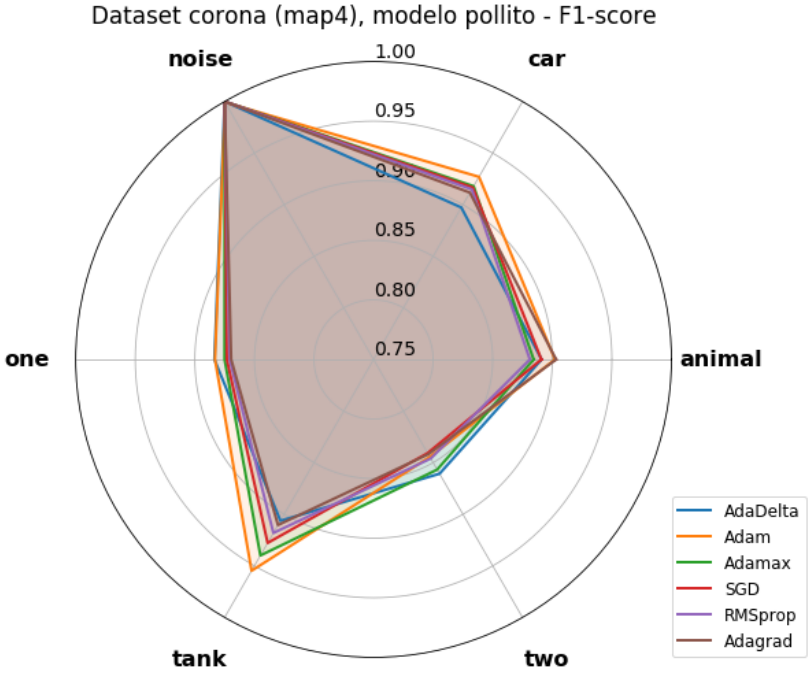
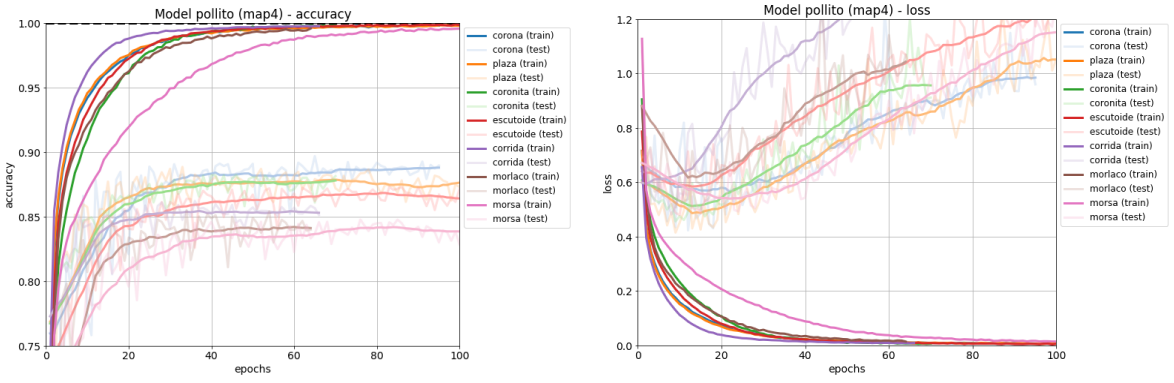


Figura 8.14: Comparación del desempeño por clase entre los optimizadores evaluados con el dataset *corona map4* (partición test), utilizando la métrica F1-score.



(a) accuracy **(b)** loss

Figura 8.15: Comparación del desempeño con datasets distintos utilizando el modelo *pollito* y el mapeo #4, a lo largo de las épocas de entrenamiento.

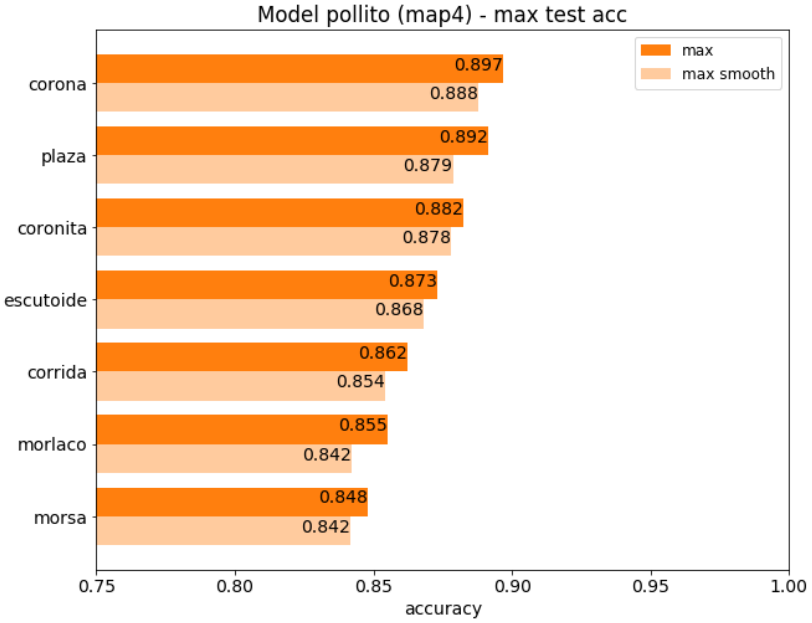


Figura 8.16: Comparación de los mejores valores de desempeño obtenidos para cada uno de los datasets evaluados utilizando el modelo *pollito* y el mapeo #4. Las barras *max smooth* se corresponden al máximo de cada curva suavizada usando una ventana de 20 muestras.

mencionados (mapeo #4), utilizando el modelo *pollito*, optimizador *AdaDelta* con igual parametrización. En la figura 8.15 puede verse que la dinámica del entrenamiento es similar en todos los casos, salvo para el dataset **morsa**, que se conformó utilizando el pre-procesamiento DCT (ver 5.5.6). En este último caso, la velocidad de convergencia fue la más lenta y el desempeño fue el menor de todos, más de 4,5 % por debajo de *corona*.

En cuanto al desempeño general, en todos los casos se consiguió más del 99 % de accuracy sobre la partición *train*, pero una dispersión mayor sobre la partición *test*. En la figura 8.16 pueden verse los valores de desempeño para la partición *test* de los distintos datasets. El dataset **corona** (utilizado como dataset por defecto) es el que mejor desempeño obtuvo. Este se conformó usando el espectrograma clásico y surgió como evolución del dataset *escutoide*. Para el dataset **escutoide** se utilizó menor rango dinámico (70 *dB* frente a los 100 *dB* del dataset *corona*) lo que genera espectrogramas menos ruidosos pero con pérdida de información de componentes espectrales débiles. También tiene un mayor *frame overlap* para data-augmentation (3,5 frente a 1,5 segundos) lo que genera muestras con menos variedad de distorsiones. Entre estos datasets, escutoide logró 2 % menos de accuracy, que estaría dependiendo fuertemente del rango dinámico (esto se sustentará más adelante).

El dataset **coronita** es básicamente el dataset *corona* sin data-augmentation. Notar que la incorporación de data-augmentation permitió incrementar cerca del 1,5 % el accuracy. Esto respalda el hecho de que la diferencia entre *escutoide* y *corona* no se

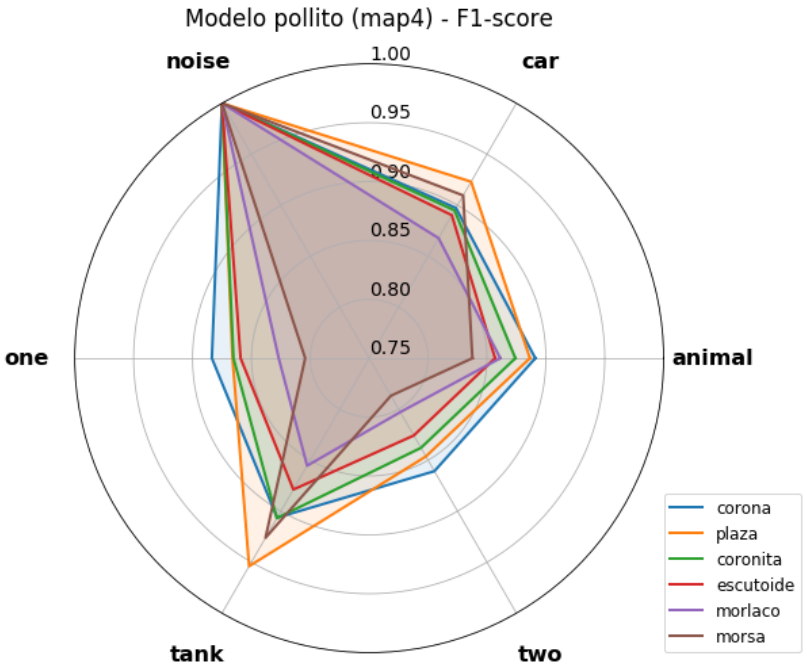


Figura 8.17: Comparación del desempeño (partición test) por clase entre los datasets evaluados utilizando el modelo *pollito* y el mapeo #4, utilizando la métrica F1-score.

deba fuertemente a la parametrización del data-augmentation sino al rango dinámico. Como trabajo futuro podrían conformarse datasets barriendo parámetros de rango dinámico para analizar la sensibilidad a este parámetro.

El dataset **corrida** también es una variación del dataset *escutoide* en donde se buscó incrementar la cantidad de muestras de train y de test incrementando el frame overlap para obtener más frames de los segmentos válidos; pero así también se redujo 5 dB el *segmentation threshold* y se relajaron levemente los parámetros *max gap time* y *min segment length* para descartar menos muestras de las adquisiciones. Esta estrategia no trajo beneficios, ya que el dataset *corrida* consiguió 3,5 % menos accuracy aproximadamente. Esto enfatiza el hecho de que la elección de los segmentos útiles para entrenamiento y evaluación de desempeño tiene un impacto significativo en el desempeño del clasificador.

El dataset **plaza** es una variación del dataset *corona*, pero usando el pre-procesamiento *multibanks* (base 6), que genera en el espectrograma una expansión de las bajas frecuencias a costa de las altas. El desempeño máximo obtenido es muy similar al *corona*, con una media levemente menor. Si bien demuestra que podría ser útil seguir explorando parametrizaciones de este dataset, hasta ahora no justifica utilizar un pre-procesamiento más demandante de capacidad de cálculo.

El dataset **morlaco** se conformó utilizando un escalograma (ver 5.5.4) en lugar de un espectrograma. Para este caso se utilizó la wavelet *Complex Morlet* con $B = 1$ y $C = 1{,}5$. Con este dataset se consiguió un poco menos del 4 % de accuracy respecto

de *corona*, por lo que su desempeño no ha sido muy satisfactorio. Sin embargo, el uso de wavelets parece prometedor debido a sus características para resolver el compromiso entre resolución frecuencial y temporal. Es recomendable, entonces, evaluar más datasets utilizando otras wavelets y parametrizaciones de las mismas. [26]

En la figura 8.17 puede verse el desempeño por clase de los distintos datasets. Aquí no hay ningún patrón relevante que notar, más allá de aquel que, a menor desempeño general, menos desempeño se obtiene en las clases *one* y *two* en relación al resto, como ya se ha notado en otros análisis.

Es importante realizar la observación de que estos resultados son obtenidos utilizando un modelo (*pollito*) que ha sido seleccionado como relevante en función de los resultados que se fueron obteniendo durante el desarrollo para los datasets basados en espectrogramas, más que nada el dataset *escutoide*. Entonces, aquellos datasets que utilizan otras técnicas de pre-procesamiento podrían llegar a funcionar mejor con otros modelos. Notar también que los desempeños obtenidos oscilan entre el 84 % y el 90 % de accuracy para la partición de *test*, por lo que superan considerablemente el requerimiento inicial.

8.4. Pos-procesamiento

Hasta el momento se han presentado los resultados a nivel muestras, o sea, a nivel *frame*. En la práctica, una vez implementado el clasificador en la plataforma radar, habrá una secuencia (stream) de frames que se entregarán al clasificador por cada blanco detectado y sobre el que se haga seguimiento. Cada uno de los frames de esta secuencia generará un reporte de clasificación, por lo que a la salida del clasificador se dispondrá de una **secuencia de clasificación**. Tener en cuenta que la cantidad de frames generados por unidad de tiempo, por cada blanco, no necesariamente debe coincidir con la configuración del dataset con que fue entrenado el modelo del clasificador.

En la figura 8.18 se muestra la clasificación a lo largo de una señal, utilizando el *mapeo #4*, el modelo *pollito* y la misma parametrización de pre-procesamiento utilizada para la conformación del dataset *corona*. En este caso se eligió una señal con varios segmentos válidos y gran parte con interferencias o falta de señal. Puede observarse que la mayor parte los reportes válidos (sombreados en gris) de clasificación de los frames de segmentos válidos (sombreado en celeste), el clasificador entrega una clasificación correcta. Este proceso de identificación de segmentos válidos permite, entonces, quedarnos con la porción de la clasificación que se considera como relevante y evitar elevar la tasa de falsas alarmas debido a problemas de la señal Doppler. Este proceso puede considerarse como uno de los tipos de pos-procesamientos mencionados

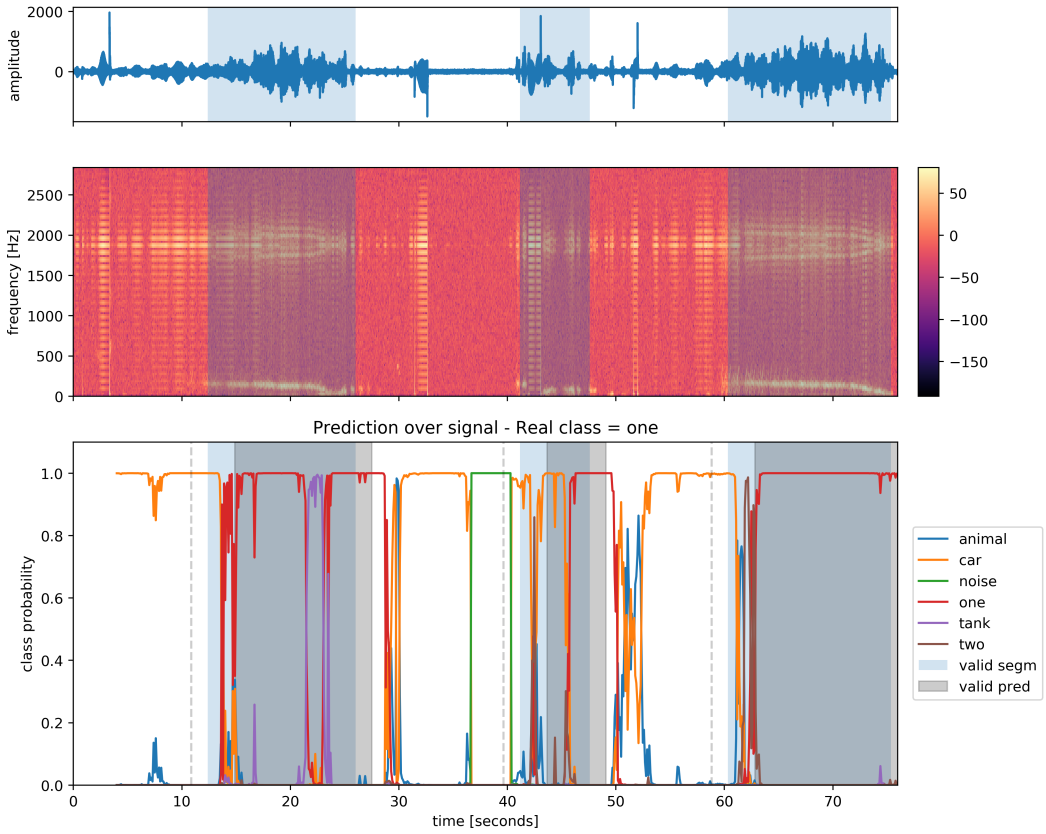
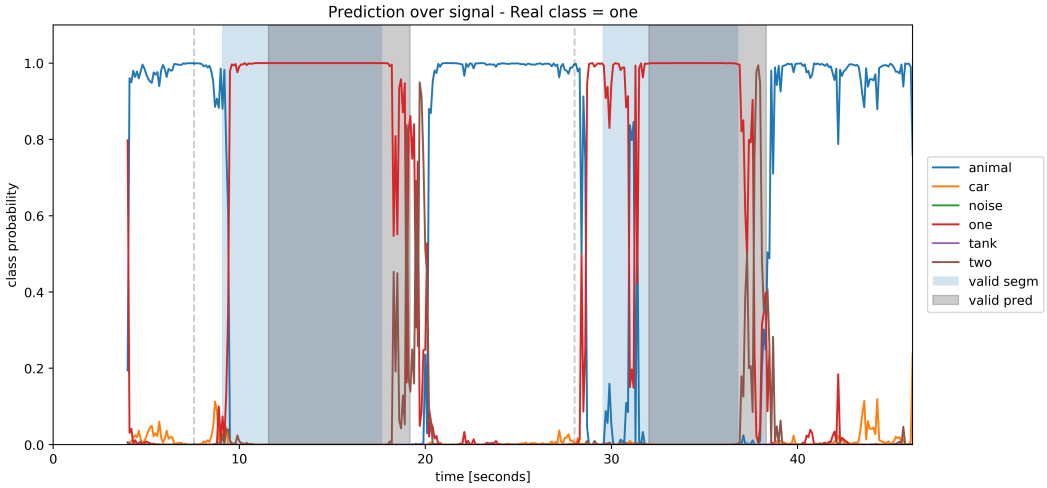


Figura 8.18: Predicciones sobre la señal Doppler de la muestra *person-one-0-run-go_away-26_18* usando un *frame stride* de 0,1 segundos (paso de cada muestra). Los segmentos válidos se resaltan en celeste, mientras que las clasificaciones válidas se resaltan en gris (se toma como válida la clasificación de un frame que contenga al menos 2,5 segundos de un segmento válido).

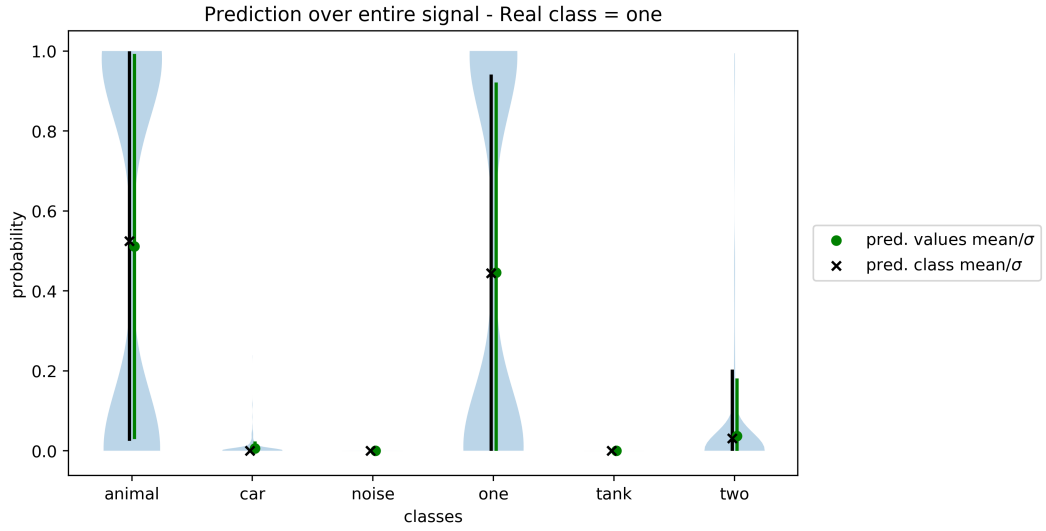
preliminarmente en 7.7.

Podría usarse el historial de clasificación sobre ese blanco para ir ajustando los porcentajes de probabilidad asignada a cada clase para ese blanco puntual, o bien, si el reporte tiene que ser simplificado, simplemente ajustar qué clase se le asigna a dicho blanco luego de un tiempo determinado de adquisición. Raramente el usuario visualizará la evolución del reporte para uno o más blancos. Un tipo de reporte no tan simplificado podría ser un gráfico de violines, como se muestran en las figuras 8.19b y 8.20b. En estas figuras se muestra un caso de una señal problemática (perteneciente a la partición de *test*), que si se considerara toda la secuencia de clasificación, la clase más probable sería *animal* (con algo más del 50 %), mientras que la clase real es *one* (alrededor de 45 %), ver figura 8.19. Sin embargo, en este último caso, usando la información de las distribuciones estadísticas, el usuario podría inferir que la clasificación como *animal* es dudosa y que la segunda probable es *one*; situación que no podría visualizarse con un ícono sobre un mapa.

Si aplicamos el filtrado por potencia de señal (selección de segmentos útiles por threshold en la densidad espectral de potencia), la predicción sobre la adquisición

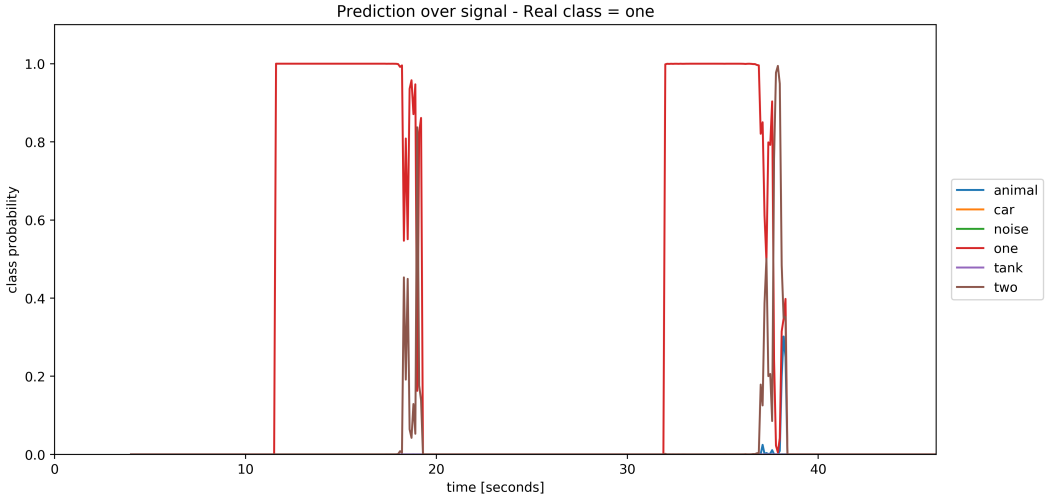


(a) tiempo

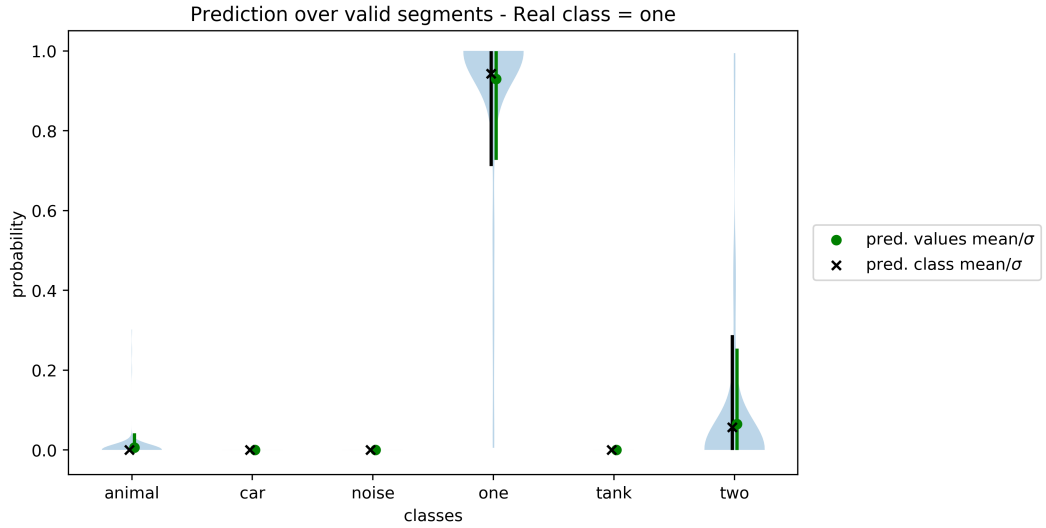


(b) violines

Figura 8.19: Reporte de clasificación sin filtrar (con los intervalos de tiempo identificados), correspondiente a la adquisición *person-one-0-normal-zigzag-2_9*. En (a) se muestra el reporte en función del tiempo de adquisición. En (b) se muestra el reporte en formato violín para cada clase, donde lo verde indica la estadística (media, desviación estándar) sobre los valores de probabilidades a la salida del clasificador, mientras que en negro se indica la estadística sobre la clase más probable (ocurrencia) frame a frame.



(a) tiempo



(b) violines

Figura 8.20: Reporte de clasificación filtrado, correspondiente a la adquisición *person-one-0-normal-zigzag-2_9*. En (a) se muestra el reporte en función del tiempo de adquisición. En (b) se muestra el reporte en formato violín para cada clase, donde lo verde indica la estadística (media, desviación estándar) sobre los valores de probabilidades a la salida del clasificador, mientras que en negro se indica la estadística sobre la clase más probable (ocurrencia) frame a frame.

puede refinarse mostrando resultados muy diferentes, como los que se pueden ver en la figura 8.20 para la misma señal que se utilizó anteriormente (figura 8.19). Ahora, la probabilidad de que el blanco pertenezca a la clase *one* es superior al 95 %, y la segunda más probable ahora es *two*. Esto demuestra que el pos-procesamiento es necesario para mejorar la calidad del producto que entrega el clasificador y que el filtrado por potencia es un método simple, con baja demanda computacional, que mejora sustancialmente dicha calidad.

8.4.1. Predicciones sobre adquisiciones

Utilizando el pos-procesamiento podemos presentar los resultados de desempeño del clasificador en términos de las adquisiciones completas y no frame a frame de cada una de ellas. En general, en trabajos similares suele evaluarse el desempeño de esta manera, lo que resulta más representativo de la calidad del producto entregado al usuario, pero deja fuera otros aspectos importantes como falsas alarmas temporales o intermitencias en la clasificación.

Un abordaje conservador del análisis de desempeño sobre las señales completas es utilizar todos los reportes de clasificación sin importar si pertenecen a un segmento válido, infiriendo la clase a la que pertenece esa adquisición tomando la media por clase de todos los reportes y eligiendo la media mayor. Los resultados de este análisis se muestran en la columna *entire signal* de la tabla 8.2. En la misma tabla también se muestran los resultados al considerar solamente los reportes de clasificación de los segmentos considerados válidos, ver columna *valid segments* de la misma tabla.

	entire signal	*valid segments*	*num of signals*
Train performance	98.29 %	99.43 %	351
Test performance	94.04 %	94.04 %	218
Overall performance	96.66 %	97.36 %	569

Tabla 8.2: Desempeño del pos-procesamiento de las predicciones sobre cada una de las señales de las adquisiciones.

La cantidad de señales en cada partición se corresponde con lo obtenido en el proceso de conformación del dataset *corona map #4*. Dentro de esas cantidades, 80 señales fueron generadas sintéticamente como señales de ruido (clase *noise*), donde 56 están en la partición *train* y 24 en la partición *test*.

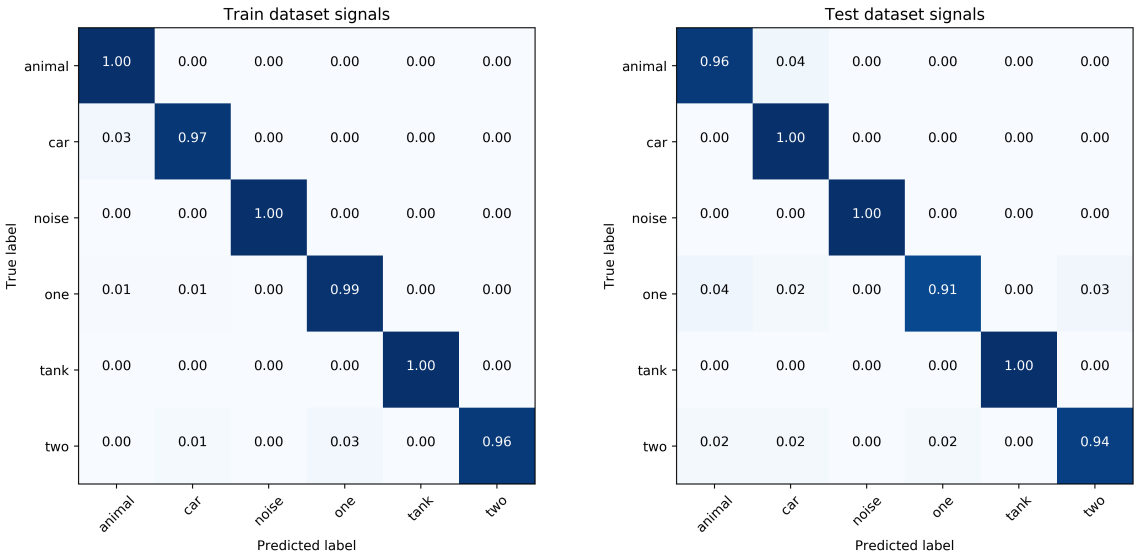
Puede concluirse a partir de las métricas generales mostradas, que el desempeño

del clasificador al entregar los reportes sobre las adquisiciones, y no frame a frame, incrementa considerablemente su desempeño. Notar que este mismo modelo mostraba un desempeño del 89.7 % sobre la partición de *test* (estadística frame a frame) y que al clasificar sobre cada adquisición, su desempeño subió 7 %.

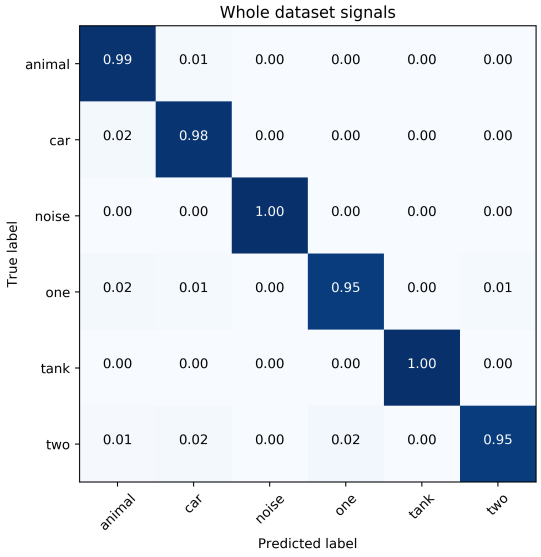
Las matrices de confusión correspondientes a la clasificación sobre las señales (*entire signal*) se encuentran en la figura 8.21. Se puede observar que los valores obtenidos son todos altamente satisfactorios para una aplicación de este tipo. Notar en particular la mejora para la clase *two* en el dataset de test. No hay que perder de vista que en estos resultados influye la cantidad de adquisiciones por clase, el largo de las mismas y la calidad de cada una (al no filtrarse los reportes), tal como se analizó en la sección 4.5.

A continuación se deja un listado de las señales que fueron clasificadas erróneamente:

```
@test dataset : /SPL/animal/safari/28_18.mat
@test dataset : /SPL/person/one/0/normal/zigzag/2_9.mat
@train dataset: /SPL/person/one/0/run/26_17.mat
@test dataset : /SPL/person/one/0/run/go_away/26_18.mat
@test dataset : /SPL/person/one/0/slow/1_13.mat
@test dataset : /SPL/person/one/45/normal/zanav/19_82.mat
@test dataset : /SPL/person/one/45/normal/zigzag/4_18.mat
@train dataset: /SPL/person/one/45/slow/4_8.mat
@test dataset : /SPL/person/one/60/fast/3_32.mat
@test dataset : /SPL/person/one/60/normal/strong_hands/19_53.mat
@test dataset : /SPL/person/one/60/run/19_58.mat
@test dataset : /SPL/person/one/60/run/22_99.mat
@test dataset : /SPL/person/two/0/fast/syncrony/19_1.mat
@train dataset: /SPL/person/two/0/normal/1_50.mat
@train dataset: /SPL/person/two/0/normal/go_away/21_24.mat
@train dataset: /SPL/person/two/0/normal/zigzag/1_73.mat
@test dataset : /SPL/person/two/30/normal/21_38.mat
@test dataset : /SPL/person/two/30/normal/line/2_27.mat
@train dataset: /SPL/vehicle/car/cars_on_the_road/10_32.mat
```



(a) train **(b)** test



(c) dataset completo

Figura 8.21: Matrices de confusión de la clasificación realizada sobre las adquisiciones del dataset SPL, sin filtrar los reportes por segmentos válidos. Se utilizó el modelo *pollito* y el preprocesamiento definido para el dataset *corona map #4*, como así también su partición train/test.

Parte III

Conclusiones

Sobre el desarrollo y resultados

La conclusión general sobre el desarrollo realizado es que **se ha logrado implementar un Clasificador de Blancos Radar**, en este caso aplicable a un conjunto de blancos terrestres, utilizando Redes Neuronales Convolucionales; que cumple con los objetivos iniciales definidos para el proyecto, detallados en 1.2. Es conveniente, entonces, realizar un repaso de dichos objetivos para poder justificar el cumplimiento de cada uno de ellos mediante los resultados obtenidos:

- *Implementar un clasificador de blancos radar*

  Se ha probado que toda la cadena de clasificación diseñada (ver capítulo 3) y entrenada ha logrado discriminar las firmas micro-Doppler de diversos blancos radar terrestres, tales como: una o dos personas, automóviles, tanques, animales varios.

  La estrategia de extraer frames de la señal Doppler para construir imágenes a partir de los correspondientes espectrogramas ha sido adecuada para la aplicación (ver capítulo 5). Esto mismo ha permitido construir muchas más muestras que el número de adquisiciones radar disponibles, dando paso a implementar una cadena de clasificación útil para la clasificación en tiempo real de firmas micro-Doppler de blancos.

- *Implementar el clasificador utilizando técnicas y algoritmos de Deep Learning*

  Dentro de la diversidad algoritmos de *Deep Learning*, esta tesis se ha concentrado sólo en los del tipo *Convolutional Neural Networks* (ver capítulo 6), de manera que este objetivo se verifica simplemente por inspección de los modelos propuestos (ver 6.9).

  El uso de este tipo de algoritmos ha sido acompañado por un estudio sobre la sensibilidad de los mismos a los múltiples hiper-parámetros (ya sea de los modelos propiamente dichos o de las cadenas presentadas) y técnicas para minimizar distintos problemas tales como: error de generalización, sesgos a nivel clase o sobre-ajuste (overfitting). Entonces, más allá de la utilización de este tipo de algoritmos, se ha realizado un estudio detallado de los mismos para entender su funcionamiento y los métodos de optimización.

- Clasificador que pueda implementarse en el hardware de procesamiento disponible en los radares, con uso bajo a moderado de los recursos de dicho hardware

Este objetivo hace foco en la factibilidad de implementación, donde la manera más adecuada de verificarse sería realizando una implementación en alguna plataforma radar. Como la implementación en este tipo de plataformas ha quedado fuera del alcance de la tesis, el abordaje para el desarrollo fue implementar modelos con arquitecturas sencillas y una baja cantidad de parámetros (pesos). Desde este punto de vista, los modelos *granadero* y *pollex* tienen muy pocos pesos (alrededor de 700 mil, ver 6.9) en comparación con CNN convencionales (mayor a 10 millones, generalmente).

Otro aspecto que soporta esta estrategia es que se han utilizado librerías ampliamente utilizadas en la industria, de código abierto y portables en distintos tipos de plataformas de procesamiento, las cuales son utilizadas en las plataformas radar, que hacen uso de procesadores del tipo GPGPU o Field-Programmable Gate Array (FPGA).

- Conseguir un desempeño en precisión de al menos 80 % para todas las clases, o comparable con el obtenido con otras técnicas, para el mismo dataset

Los modelos que se han propuesto, entrenado y evaluado, tales como *pollito* y *granadero* (junto a sus variantes) han superado ampliamente el desempeño esperado (80 % de precisión para todas las clases), tanto a nivel general como a nivel clases, ver capítulo 8.

Para el caso puntual del modelo *pollito*, que presentó el mejor desempeño, se obtuvo una *accuracy* general medio de 88,8 % para la partición de *test* (ver 8.1). Si detallamos el desempeño por clase, el modelo presentó su menor desempeño para la clase *two* (dos personas), con un 84 % de *accuracy*; mayor al 89 % para el resto, presentando su mejor desempeño para las clases *tank* con 99 % y *noise* con el 100 % (ver 8.1.1).

Este desempeño pudo mejorarse aún más utilizando pos-procesamiento de los reportes del clasificador a lo largo de las distintas adquisiciones utilizadas para la conformación del dataset (ver 8.4). En este caso, la clase con menor desempeño para la partición de *test* fue *one* (una persona) con un 91 % de *accuracy*, lográndose una mejora significativa sobre el caso en donde no se utiliza pos-procesamiento.

- *Entrenar y validar el clasificador utilizando un dataset con datos reales de un radar*

Los modelos han sido entrenados y evaluados usando sólo datos reales de un radar, particularmente adquisiciones (crudas) del *SPL dataset* [39], cuyas características se detallan en el capítulo 4.

Estas adquisiciones se pre-procesaron para conformar distintos datasets con los que los modelos fueron entrenados y evaluados, tal como se detalla en el capítulo 5. De esta manera, el cumplimiento del objetivo se verifica por inspección.

- *Latencia para la clasificación de un blanco menor o igual a 4 segundos*

De la manera que fue implementada la cadena de clasificación (ver 3.1)), la latencia inicial de clasificación sobre la señal Doppler de un blanco viene dada por el tiempo necesario para conformar el primer *frame* más el tiempo de inferencia del mismo. En todos los datasets que se conformaron se utilizó una duración de *frame* de 4 segundos, y debido a que los tiempos de pre-procesamiento e inferencias no son significativos (en 8.1.4 se muestra que la suma de ambos es menor a 23 ms), el objetivo se verifica por diseño.

Esta latencia puede reducirse de distintas maneras. Una de ella es conformando el primer *frame* con una porción de ruido inicial (previo a la primera muestra de la señal Doppler del blanco) y completando el resto con la señal Doppler. Otra manera es reduciendo el tiempo de cada *frame* al valor de latencia deseado. En ambos casos se esperaría que el desempeño se vea afectado, pero en el primer caso, el desempeño se vería afectado sólo en los primeros *frames* (los que contienen ruido).

- *Tiempo de clasificación menor o igual a 100 ms (capacidad de clasificar como mínimo 10 blancos diferentes por segundo)*

El tiempo total de clasificación, para el modelo *pollito* (el de mayor capacidad entre los propuestos) es menor a 23 ms (ver 8.1.4), por lo que se verifica el cumplimiento de este objetivo.

Este tiempo ha sido medido utilizando una plataforma particular (que usa una GPGPU) con una menor capacidad de cálculo que las plataformas que se utilizan en radares de alta performance, pero comparable o mayor que las plataformas utilizadas en radares portátiles (o equivalentes). Sin embargo, se ha identificado que una gran porción del tiempo de clasificación es debido al movimiento de datos entre la CPU y la GPGPU de la plataforma, de manera que si el pre-procesamiento y clasificación se realizara en un mismo procesador, los tiempos se reducirían significativamente.

A continuación se realizan algunas observaciones importantes obtenidas a lo largo del desarrollo de la tesis y en consecuencia de los resultados obtenidos:

Datos crudos y datasets

El dataset conseguido tiene las señales crudas de las adquisiciones de las señales Doppler, presentando gran cantidad de segmentos defectuosos, sin señal o con interferencias; fue necesario dedicar gran parte del desarrollo a implementar una cadena de procesamiento que lograra extraer y etiquetar los segmentos útiles automáticamente, para no realizar esa tarea manualmente. Esta cadena, denominada *Dataset Chain*, permitió a su vez incorporar fácilmente diversas etapas de procesamiento, lo que derivó en poder experimentar con distintas parametrizaciones y técnicas de pre-procesamiento. Todo ello dio lugar a la conformación de distintos datasets, que conforman una base de referencia para la evaluación de los modelos de clasificación; y que hoy quedan como referencia para futuros trabajos relacionados. Más allá de que el clasificador muestra un buen desempeño, muchos de los problemas relacionados con la calidad del dataset afectan en gran medida al desempeño; es por ello que **es aconsejable ampliar el dataset actual con muestras de mejor calidad** y de las cuáles se tenga mayor información de cómo fueron adquiridas. Sería aún más conveniente que estas nuevas muestras sean adquiridas con distintos tipos de radares, en distintos escenarios e inclusive generadas sintéticamente utilizando modelos de blancos y clutter.

El desarrollo de la *Dataset Chain* se traslada directamente a la cadena de clasificación que se instancia en el radar, permitiendo identificar segmentos válidos en tiempo real, lo que permite mejorar aún más la calidad del producto entregado por el radar, al identificar o filtrar aquellas predicciones realizadas sobre porciones de la señal Doppler que no sean fiables.

Convertir las señales (de audio) Doppler a imágenes, mediante el cálculo del *espectrograma* de las mismas, fue la idea inicial para el trabajo, motivada mayormente por trabajo previos para clasificación de señales de audio de la vida diaria. Sin embargo se amplió esa idea incluyendo otros tipos de procesamiento como *escalogramas* (uso de *wavelets*), transformaciones utilizadas para compresión de imágenes y algunas otras variantes del espectrograma. Al menos, con los modelos evaluados y las parametrizaciones utilizadas para estos pre-procesamientos, no se ha logrado superar el desempeño del clasificador al utilizar el espectrograma convencional; pero el desempeño de las otras técnicas ha sido suficiente para cumplir los objetivos igualmente. Esto deja abierta la puerta para experimentar más en profundidad con algunas de ellas. El ajuste de los parámetros para el cálculo del espectrograma, como el largo del ventaneo, el solapa-

miento, cantidad de muestras para la FFT, entre otras; resulta fundamental para lograr que las componentes frecuenciales se puedan discriminar mejor en la imagen resultante. También se observó que el desempeño muestra una sensibilidad relevante al rango dinámico de las señales.

El mapeo que se utilizó con mayor frecuencia en los experimentos, permitió mostrar que el clasificador tiene un buen desempeño en la discriminación de *autos*, *tanques*, *animal*es y *personas*; e inclusive, discriminar cuando en la celda de resolución hay más de una persona. Es esperable entonces que al conjugar estas clases en clases aún más genéricas como *vehículo* y *persona*, se logren mejores resultados. Sería deseable igualmente expandir el dataset con muestras de otro tipo de blancos radar, como ser blancos aéreos: *helicópteros* y *drones* por citar algunos; y con muestras de distintos tipos de *clutter*.

Aumentar la cantidad de datos a partir de distorsiones de las señales originales (Data-Augmentation) mejoró levemente el desempeño de los modelos, pero sería recomendable incorporar otros tipos de distorsiones y explorar con distintas parametrizaciones para evaluar su impacto.

Modelos

Durante la etapa de desarrollo de los modelos, se han logrado cumplir los objetivos de desempeño utilizando modelos CNN de mucha menor complejidad que los modelos utilizados para la clasificación de imágenes de la vida diaria. Esto permite desplegar los modelos en plataformas con menor capacidad de cómputo, o bien, implementar varias cadenas de clasificación en paralelo para incrementar la cantidad de predicciones por segundo que podría realizar el radar; más aún si realiza el tracking sobre múltiples blancos de manera simultánea. A su vez, se mostró que los modelos CNN evaluados superan el desempeño de redes neuronales convencionales, lo que justifica la línea de investigación sobre la clasificación de espectrogramas de señales Doppler utilizando DL.

Se observó que la arquitectura de la etapa de *features learning* y su dimensionamiento, influyó más en el desempeño de los modelos que la etapa de *classification*. En función de esto, sería conveniente seguir explorando arquitecturas, con el foco puesto en esa primera etapa, para buscar disminuir el error de generalización. Si en un futuro existe la posibilidad de trabajar con un dataset de mejor calidad, este será uno de los aspectos importantes a tener en cuenta para obtener un buen desempeño sin necesidad de construir modelos de gran capacidad.

Distintos optimizadores fueron estudiados y utilizados durante los diversos experimentos. Todos ellos han mostrado un comportamiento satisfactorio en cuanto al desempeño alcanzado y al tiempo de convergencia; pero los del tipo adaptivo han demandado

ajuste fino menor para lograrlo. En general las épocas para la convergencia y el desempeño sobre la partición *train* han sido muy buenas, lo que no ha exigido trabajar mucho sobre el tema optimizadores, más aún que no brindaron diferencias significativas en el desempeño sobre la partición *test*.

El ajuste de los distintos hiper-parámetros, como se ha mencionado a lo largo del documento, ha sido una tarea que se fue realizando manualmente en función de los análisis de sensibilidad que se iban realizando. En el proceso de entrenamiento de modelos de DL, es una práctica común realizar *Hyperparameters Optimization* (optimización de los hiper-parámetros) automática. Con el grado de madurez alcanzado sobre la línea de desarrollo abordada con este trabajo, parece conveniente abordar los próximos experimentos y análisis agregando esta funcionalidad al entorno de desarrollo.

Finalmente, el pos-procesamiento de los reportes generados por el clasificador frame a frame ha mostrado un incremento del desempeño al mirar la clasificación sobre la señal completa y no sobre cada frame; consiguiendo un producto de mejor calidad y más útil para el usuario.

Entorno de desarrollo y herramientas

Mucho tiempo se ha dedicado a la escritura de código para conformar un entorno de desarrollo y herramientas que facilitaran la implementación de las cadenas funcionales descriptas, definición y compilación de los modelos, parametrizaciones vía archivos de configuración, visualización de resultados y datos de puntos de inspección. Esto fue muy beneficioso para poder realizar una enorme cantidad de experimentos y poder mantener la trazabilidad de las configuraciones y los datos obtenidos, generando un ahorro enorme de tiempo, disponer de experimentos repetibles y datos confiables. A su vez, estas cadenas pueden utilizarse casi directamente para las implementaciones finales en la plataforma radar. Si bien hoy existen múltiples plataformas pensadas para acelerar los tiempos de desarrollo, éstas no eran muy conocidas, ni tenían un buen grado de madurez, cuando este proyecto comenzó. Sin embargo, el hecho de tener más conocimiento y control del entorno de desarrollo permite una mejor migración hacia otras plataformas, particularmente cuando no son plataformas estándares, como sería el caso de una plataforma de procesamiento radar.

El desarrollo fue realizado enteramente con software y librerías libres, de código abierto. De esta manera, no hay problemas o trabas que pueden aparecer por licenciamientos.

Futuros trabajos

A continuación se proponen algunas líneas de trabajo que parecen interesantes y promisorias, como continuación de este trabajo:

Dataset : Es importante contar con un dataset de mejor calidad, mayor cantidad de muestras y clases más diversas. Sería conveniente incorporar muestras de distintos tipos/modelos de radar, como así también datos sintéticos generados a partir de modelos de blancos y clutter. El trabajo en esta línea se concentraría en campañas de adquisiciones de datos, como así también en la formulación e implementación de modelos para datos sintéticos. Sería adecuado complementar este trabajo con análisis comparativos entre datos reales y los generados por los modelos de blancos. Parece muy conveniente incorporar drones como blancos, [104].

Análisis de sensibilidad : Como se ha mencionado anteriormente en las conclusiones, sería interesante realizar más análisis de sensibilidad en el desempeño a los distintos hiper-parámetros de las cadenas de procesamiento y de los modelos. Donde la implementación de técnicas para optimización de los hiper-parámetros sería muy recomendable en este abordaje.

Pre-procesamiento : Claramente este trabajo no profundizó sobre los distintos tipos de pre-procesamientos con los que se experimentó, ya que no era la línea principal del mismo. Como los resultados obtenidos para algunos de ellos son igualmente buenos, sería recomendable estudiarlos más en profundidad, buscando modelos con arquitecturas más adecuadas y parametrizaciones para optimizar su desempeño. El uso de *wavelets* parece muy promisorio debido a sus características y que son ampliamente usadas para el procesamiento de este tipo de señales. [32, 33, 52, 105]

Modelos estándares - Transfer Learning : Un abordaje recomendado en las aplicaciones de clasificación de imágenes es utilizar modelos "estándares" (modelos que son conocidos por su buen desempeño en este tipo de aplicaciones). El entrenamiento de los mismos no suele hacerse desde cero, sino partiendo de los pesos del modelo entrenado para otro dataset, esto último se denomina *transfer learning* (transferencia de aprendizaje). El trabajo propuesto sería evaluar estos modelos para la clasificación de los espectrogramas, y considerar utilizar *transfer learning* para agilizar el entrenamiento. [36]

Detección de múltiples patrones simultáneos : Utilizar arquitecturas y técnicas para la detección de múltiples objetos dentro de una imagen, para detectar patrones de firma micro-Doppler de múltiples blancos en la misma celda de resolución radar, o cuando se conforma la señal Doppler utilizando varias celdas a la vez. De esta manera, el clasificador identifica porciones de la imagen en donde hay un patrón posible y la clase a la que pertenecería. [24, 106]

Redes Recurrentes : Utilizar otro tipo de modelos, como por ejemplo RNN y especialmente las del tipo Long Short-Term Memory (LSTM). En este caso, el cambio de tipo de modelo puede imponer un tipo de pre-procesamiento diferente. El tipo de modelo RNN se utiliza generalmente para la clasificación de secuencias de datos, por lo que podría realizarse la clasificación de la señal Doppler directamente desde el dominio del tiempo. [53, 107]

Plataformas : La etapa que sigue a este trabajo es la implementación de la cadena de clasificación en la plataforma radar, que en general es una GPGPU con gran capacidad de cómputo. Sin embargo, hay radares de menor costo/tamaño que hacen uso de otro tipo de plataformas, menos potentes. En esa línea se puede trabajar sobre la implementación de la cadena de clasificación en dispositivos como System on Chip (SoC), FPGA o GPGPU destinadas a sistemas embebidos.

Términos especiales

chirp Señal modulada en frecuencia, tal que la frecuencia cambie de forma lineal con el tiempo.

clutter Término usado para los elementos que generan ecos de la señal radar que no son de interés. Típicamente, se refiere a elementos como la tierra, mar, lluvia.

dataset Conjunto de datos. Se denomina así al conjunto de muestras utilizadas para el entrenamiento, testeo y validación de los modelos de ML.

features Características cuantificables o enumerables que permite clasificar una muestra.

Acrónimos

P_d	Probabilidad de detección.
ADC	Analog-to-Digital Conversion/Converter (Conversión/Conversor Analógico-Digital).
AI	Artificial Intelligence (Inteligencia Artificial).
ANN	Artificial Neural Network (Red Neuronal Artificial).
ATR	Radar Automatic Target Recognition (Reconocimiento Automático de Blanco Radar).
BB	Base-Band (Banda Base).
CNN	Convolutional Neural Network (Redes Neuronales Convolucionales).
CW	Continous Wave (Onda Continua).
CWT	Continous Wavelet Transform (Transformada de Ondita Continua).
DCT	Discrete Cosine Transform (Transformada Coseno Discreta).

DFT	Discrete Fourier Transform (Transformada de Fourier Discreta).
DL	Deep Learning (Aprendizaje Profundo).
DT	Decision Trees (Árboles de Decisión).
ELU	Exponential Linear Unit (Unidad Lineal Exponencial).
FFT	Fast Fourier Transform (Transformada de Fourier Rápida).
FMCW	Frequency Modulated Continous Wave (Onda Continua con Frecuencia Modulada).
FPGA	Field-Programmable Gate Array.
GPGPU	General Purpose Graphical Processing Unit (Unidad de Procesamiento Gráfico de Propósito General).
HPA	High Power Amplifier (Amplificador de Potencia).
HRR	High Range Resolution (Alta Resolución en Rango).
IF	Intermediate Frequency (Frecuencia Intermedia).
ISAR	Inverse SAR (SAR Inverso).

JSON	JavaScript Object Notation (Notación de Objeto de JavaScript).
k-NN	k-Nearest Neighbors (k Vecinos más Cercanos).
KLD	Kullback-Leibler Divergence (Divergencia de Kullback-Leibler).
LNA	Low Noise Amplifier (Amplificador de Bajo Ruido).
LR	Linear Regression (Regresión Lineal).
LSTM	Long Short-Term Memory.
MAE	Mean Absolute Error (Error Absoluto Medio).
MFCC	Mel-Frequency Cepstral Coefficients (Coeficientes Ceptrales en las Frecuencias de Mel).
ML	Machine Learning (Aprendizaje Automático).
MSE	Mean Squared Error (Error Cuadrático Medio).
MSLE	Mean Squared Logarithmic Error (Error Logarítmico Cuadrático Medio).
NN	Neural Network (Red Neuronal).

PRF Pulse Repetition Frequency (Frecuencia de Repetición de Pulsos).

PRI Pulse Repetition Interval (Intervalo de Repetición de Pulso).

RCS Radar Cross-Section (Sección Eficaz Radar).

ReLU Rectified Linear Unit (Unidad Lineal Rectificada).

RF Radio Frecuencia(s).

RMSE Root Mean Squared Error (Raíz del Error Cuadrático Medio).

RNN Recurrent Neural Network (Redes Neuronales Recurrentes).

ROC Receiver Operating Characteristic (Característica de Operación del Receptor).

Rx Recepción.

SAR Synthetic Aperture Radar (Radar de Apertura Sintética).

BGD Batch Gradient Descent (Descenso por Gradiente por Batch).

SGD Stochastic Gradient Descent (Descenso por Gradiente Estocástico).

SINR Signal to Interferences and Noise Ratio.

SNR Signal to Noise Ratio.

SoC System on Chip.

STFT	Short-Time Fourier Transform (Transformada de Fourier de Tiempo Corto).
SVM	Support-Vector Machines (Máquinas de Vector de Soporte).
Tx	Transmisión.

Índice de figuras

Índice de tablas

Bibliografía

[1] Goodfellow, I., Bengio, Y., Courville, A. Deep Learning. MIT Press, 2016. http://www.deeplearningbook.org. xv, 93, 97, 105, 107, 108, 119, 126, 128, 129, 130, 131, 132, 133, 135

[2] David Blacknell, H. D. G. Radar Automatic Target Recognition (ATR) and Non-Cooperative Target Recognition (NCTR), cap. 2. IET Radar, Sonar and Navigation Series 33. The Institution of Engineering and Technology, 2013. 1

[3] Carmine Clemente, A. B. Developments in target micro-doppler signatures analysis: radar imaging, ultrasound and through-the-wall radar. *EURASIP Journal on Advances in Signal Processing*, **2013**, 47, 2013. 1

[4] Li, Y., Du, L., Liu, H. Moving vehicle classification based on micro-doppler signature. En: 2011 IEEE International Conference on Signal Processing, Communications and Computing (ICSPCC). IEEE, 2011.

[5] Li, Y., Du, L., Liu, H. Noise robust classification of moving vehicles via micro-doppler signatures. En: 2013 IEEE Radar Conference (RadarCon13). IEEE, 2013. 1

[6] Stove, A., Sykes, S. A doppler-based automatic target classifier for a battlefield surveillance radar. En: RADAR 2002. IEE, 2002. 2

[7] Bar-Hillel, A., Bilik, I., Hecht, R. Naive bayes nearest neighbor classification of ground moving targets. En: 2013 IEEE Radar Conference (RadarCon13). IEEE, 2013. 2

[8] Fogle, B. D., O. Ryan; Rigling. Micro-range/micro-doppler decomposition of human radar signatures. *IEEE Transactions on Aerospace and Electronic Systems*, **48**, 3058–3072, 2012. 17, 47

[9] Mesloub, A., Abed-Meraim, K., Belouchrani, A. Ground moving target classification based on micro-doppler signature using novel spectral information features. En: 2017 European Radar Conference (EURAD). IEEE, 2017.

[10] Bujakovic, D. M., Durovic, Z. M., Andric, M. S., Bondzulic, B. P., Simic, S. M. Expert system based on hidden markov models for recognition of radar targets. En: 2016 24th Telecommunications Forum (TELFOR). IEEE, 2016.

[11] Bilik, J., I.; Tabrikian. Radar target classification using doppler signatures of human locomotion models. *IEEE Transactions on Aerospace and Electronic Systems*, **43**, 1510–1522, 2007. 17, 47

[12] Eeden, J. d. B. R. N. W. B. E., W.D Van; Villiers. Micro-doppler radar classification of humans and animals in an operational environment. *Expert Systems with Applications*, pág. S0957417418300964, 2018. 17

[13] Bilik, J. C. A., I.; Tabrikian. Gmm-based target classification for ground surveillance doppler radar. *IEEE Transactions on Aerospace and Electronic Systems*, **42**, 267–278, 2006.

[14] Andrić, B. Z. B. K. A. D. G., Milenko; Bondžulić. Acoustic experimental data analysis of moving targets echoes observed by doppler radars. *Strojniški vestnik – Journal of Mechanical Engineering*, **58**, 386–393, 2012. URL http://doi.org/10.5545/sv-jme.2011.278.

[15] Foued, C., Ammar, M., Arezki, Y. Detection and classification of ground targets using a doppler RADAR. En: 2017 Seminar on Detection Systems Architectures and Technologies (DAT). IEEE, 2017. 2

[16] Vrckovnik, T. C. C., G.; Chung. Classification of impulse radar waveforms using neural networks. *International Journal of Neural Systems*, **5**, 23–37, 1994. 2

[17] Clemente, L. M. A. S. J. F. A., Carmine; Pallotta. A novel algorithm for radar classification based on doppler characteristics exploiting orthogonal pseudo-zernike polynomials. *IEEE Transactions on Aerospace and Electronic Systems*, **51**, 417–430, 2015.

[18] Smith, K. B. C. J., Graeme E.; Woodbridge. Radar micro-doppler signature classification using dynamic time warping. *IEEE Transactions on Aerospace and Electronic Systems*, **46**, 1078–1096, 2010.

[19] Molchanov, R. I. d. W. J. J. E. K. A. J., Pavlo; Harmanny. Classification of small uavs and birds by micro-doppler signatures. *International Journal of Microwave and Wireless Technologies*, **6**, 435–444, 2014.

[20] McConaghy, H. B. E. V. V., T.; Leung. Classification of audio radar signals using radial basis function neural networks. *IEEE Transactions on Instrumentation and Measurement*, **52**, 1771–1779, 2003.

[21] Molchanov, P., Astola, J., Egiazarian, K., Totsky, A. Classification of ground moving radar targets by using joint time-frequency analysis. En: 2012 IEEE Radar Conference. IEEE, 2012. 2

[22] Lecun, L. B. Y. H. P., Y.; Bottou. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, **86**, 2278–2324, 1998. 2, 93

[23] Krizhevsky, I. H. G. E., Alex; Sutskever. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, **60**, 84–90, 2017. 93

[24] Redmon, J., Divvala, S., Girshick, R., Farhadi, A. You only look once: Unified, real-time object detection, 2016. 2, 176

[25] Khodjet-Kesba, M. Automatic target classification based on radar backscattered ultra wide band signals. 2

[26] Serir, A., Bouhafsi, Y. Micro-doppler radar signature classification by time-frequency and time-scale analysis. En: 2012 11th International Conference on Information Science, Signal Processing and their Applications (ISSPA). IEEE, 2012. 160

[27] Kahl, S., Wilhelm-Stein, T., Hussein, H., Klinck, H., Kowerko, D., Ritter, M., *et al.* Large-scale bird sound classification using convolutional neural networks. 2017.

[28] Antich, J. L. D. Audio event classification using deep learning in an end-to-end approach. 2017. 109

[29] Takahashi, M. V. G. L., Naoya; Gygli. Aenet: Learning deep audio features for video analysis. *IEEE Transactions on Multimedia*, págs. 1–1, 2017. 109

[30] Adavanne, S., Drossos, K., Cakir, E., Virtanen, T. Stacked convolutional and recurrent neural networks for bird audio detection. En: 2017 25th European Signal Processing Conference (EUSIPCO). IEEE, 2017.

[31] Dahl, A., G.E.; Dong Yu; Li Deng; Acero. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *IEEE Transactions on Audio Speech and Language Processing*, **20**, 30–42, 2012.

[32] Chen, H., Zhang, P., Bai, H., Yuan, Q., Bao, X., Yan, Y. Deep convolutional neural network with scalogram for audio scene modeling. págs. 3304–3308. 2018. 28, 175

[33] Kizhakkel, V. Pulsed Radar Target Recognition Based on Micro-doppler Signatures Using Wavelet Analysis. Ohio State University, 2013. 28, 175

[34] Mohamed, A.-R. Deep neural network acoustic models for asr. 2014.

[35] Hadhrami, E. A., Mufti, M. A., Taha, B., Werghi, N. Ground moving radar targets classification based on spectrogram images using convolutional neural networks. En: 2018 19th International Radar Symposium (IRS). IEEE, 2018.

[36] Hadhrami, E. A., Mufti, M. A., Taha, B., Werghi, N. Transfer learning with convolutional neural networks for moving target classification with micro-doppler radar spectrograms. En: 2018 International Conference on Artificial Intelligence and Big Data (ICAIBD). IEEE, 2018. 2, 175

[37] Andric, M. S., Bondzulic, B. P., Zrnic, B. M. [ieee 2010 10th symposium on neural network applications in electrical engineering (neurel 2010) - belgrade, serbia (2010.09.23-2010.09.25)] 10th symposium on neural network applications in electrical engineering - the database of radar echoes from various targets with spectral analysis. 2010. 7

[38] The data base of radar echoes from various targets [online]., 2010. URL http://cid-3aaf3e18829259c0.skydrive.live.com/home.aspx. 7

[39] Ben Gurion University of the Negev, E., Computer Engineering Department, S. P. L. The data base of radar echoes from various targets [online]., 2013. URL http://www.ee.bgu.ac.il/~spl/, (Date last accessed 2-May-2018). Date last update: 30-Oct-2013. 7, 40, 41, 171

[40] Bilik, I., Tabrikian, J. Knowledge-Based Radar Detection, Tracking and Classification, cap. 9. Knowledge-based radar target classification, págs. 197–224. Wiley, 2007. 7, 40, 41

[41] Skolnik, M. I. Introduction to Radar Systems. 3ª edón. Tata McGraw-Hill, 2001. 11

[42] Richards, M. A. Signal Models, cap. 2. 1ª edón. McGraw-Hill, 2005. 15, 16, 19

[43] Boulic, N. M. T. D., Ronan; Thalmann. A global human walking model with real-time kinematic personification. *The Visual Computer*, **6**, 344–358, 1990. 17, 47

[44] Andrić, M., Bujaković, D., Bondzulic, B., Simic, S., Zrnic, B. Analysis of radar doppler signature from human data. *Radioengineering*, **23**, 04 2014. 17, 47

[45] Victor C. Chen, W. J. M., David Tahmoush. Radar Micro-Doppler Signatures: Processing and Applications. IET Radar, Sonar, Navigation Series 34. The Institution of Engineering and Technology, 2014. 17

[46] Chen, V. The Micro-Doppler Effect in Radar. Artech House radar library. Artech House, 2019. 17

[47] Andrić, M., Bondzulic, B., Zrnic, B. Feature extraction related to target classification for a radar doppler echoes. En: 18th Telecommunications forum, págs. 725–728. 2010. 17

[48] Garcia-Rubia, *et al.* Analysis of moving human micro-doppler signature in forest environments. *Progress In Electromagnetics Research*, **148**, 1–14, 2014.

[49] Thayaparan, T., Abrol, S., Riseborough, E. Micro-doppler radar signatures for itelligent target recognition. Defence Research and Development Canada, 2004. Technical Memorandum, Unclassified.

[50] Liang, C., Yan, L., Lu, D., Li, J., Li, Y. Simulation and analysis of micro-doppler signatures of tracked vehicles. En: IET International Radar Conference 2013. Institution of Engineering and Technology, 2013. 17, 47

[51] Richards, M. A. Signal Models, cap. 3. 1ª ed$^{\text{ón}}$. McGraw-Hill, 2005. 20

[52] Lobato, W. Wavelet multiresolution analysis and dyadic scalogram for detection of epileptiform paroxysms in electroencephalographic signals. *Research on Biomedical Engineering*, **33**, 195–201, 09 2017. 28, 175

[53] Lipton, Z. C., Berkowitz, J., Elkan, C. A critical review of recurrent neural networks for sequence learning, 2015. 28, 176

[54] Foundation, P. S. Python: A dynamic, open source programming language. URL https://www.python.org/. 31

[55] Chollet, F., *et al.* Keras. https://github.com/fchollet/keras, 2015. 31, 37, 137

[56] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., *et al.* TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/, software available from tensorflow.org. 31, 37, 137

[57] Tensorboard web site. URL https://www.tensorflow.org/tensorboard. 37

[58] Wikipedia contributors. Short-time fourier transform — Wikipedia, the free encyclopedia, 2020. URL https://en.wikipedia.org/w/index.php?title=Short-time_Fourier_transform&oldid=992533854. 54

[59] Wikipedia contributors. Discrete fourier transform — Wikipedia, the free encyclopedia, 2020. URL https://en.wikipedia.org/w/index.php?title=Discrete_Fourier_transform&oldid=995097422. 54

[60] Wikipedia contributors. Fast fourier transform — Wikipedia, the free encyclopedia, 2020. URL https://en.wikipedia.org/w/index.php?title=Fast_Fourier_transform&oldid=994477017. 54

[61] Alan V. Oppenheim, R. W. S. Discrete-Time Signal Processing. Prentice Hall Signal Processing Series, 3ª edón. Prentice Hall, 2010. 54, 55

[62] Wikipedia contributors. Spectrogram — Wikipedia, the free encyclopedia, 2020. URL https://en.wikipedia.org/w/index.php?title=Spectrogram&oldid=993696837. 55

[63] Wikipedia contributors. Cross-validation (statistics) — Wikipedia, the free encyclopedia, 2020. URL https://en.wikipedia.org/w/index.php?title=Cross-validation_(statistics)&oldid=993367784. 66

[64] Schreiber, R. E. M. M. D., Ethan L.; Korf. Optimal multi-way number partitioning. *Journal of the ACM*, **65**, 1–61, 2018. 67

[65] Strang, T. N. G. Wavelets and Filter Banks. 2ª edón. Wellesley College, 1996. 80

[66] Lilly, J. M., Olhede, S. C. On the analytic wavelet transform. *IEEE Transactions on Information Theory*, **56** (8), 4135–4156, 2010. 80

[67] LeCun, *et al.* Handwritten digit recognition: applications of neural network chips and automatic learning. *IEEE Communications Magazine*, **27**, 0–46, 1989. 93

[68] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., Fei-Fei, L. ImageNet: A large-scale hierarchical image database. En: 2009 IEEE Conference on Computer Vision and Pattern Recognition. IEEE, 2009. URL http://image-net.org/. 93

[69] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., *et al.* ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, **115** (3), 211–252, 2015. 93

[70] Simonyan, K., Zisserman, A. Very deep convolutional networks for large-scale image recognition, 2015. 94

[71] He, K., Zhang, X., Ren, S., Sun, J. Deep residual learning for image recognition, 2015. 94

[72] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., *et al.* Going deeper with convolutions, 2014. 94

[73] Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., *et al.* Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017. 94

[74] Haykin, S. S. Neural networks : a comprehensive foundation. 3ª edón. Prentice Hall, 2008. 95, 100, 105

[75] Kolmogorov, A. On the representation of continuous functions of several variables by superpositions of continuous functions of a smaller number of variables. En: Proceedings of the USSR Academy of Sciences, pág. 179–182. English translation: Amer. Math. Soc. Transl., 17 (1961), pp. 369–373. 100

[76] Stanford. Cs231n: Convolutional neural networks for visual recognition. URL https://cs231n.github.io/. 101, 117, 118, 119, 120, 122, 127, 128, 129, 133

[77] Nielsen, M. A. Neural networks and deep learning, 2015. URL http://neuralnetworksanddeeplearning.com/, determination Press. 101, 107, 108, 118, 119

[78] Zhang, C., Bengio, S., Hardt, M., Recht, B., Vinyals, O. Understanding deep learning requires rethinking generalization, 2017.

[79] Arpit, D., *et al.* A closer look at memorization in deep networks, 2017. 107

[80] Nitish Srivastava, A. K. I. S. R. S., Geoffrey Hinton. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, **15** (56), 1929–1958, 2014. 107

[81] Ioffe, S., Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, **abs/1502.03167**, 2015. URL http://arxiv.org/abs/1502.03167. 108

[82] Science/Medium, T. D. Batch normalization in neural networks. URL https://towardsdatascience.com/batch-normalization-in-neural-networks-1ac91516821c. 109

[83] Choromanska, A., Henaff, M., Mathieu, M., Arous, G. B., LeCun, Y. The loss surfaces of multilayer networks, 2015. 118

[84] Wikipedia. Gradient descent. URL https://en.wikipedia.org/wiki/Gradient_descent. 118, 119

[85] Wikipedia. Gradient. URL https://en.wikipedia.org/wiki/Gradient. 118

[86] Rumelhart, G. E. W. R. J., David E.; Hinton. Learning representations by back-propagating errors. *Nature*, **323**, 1986. 119

[87] He, K., Zhang, X., Ren, S., Sun, J. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. En: Proceedings of the IEEE International Conference on Computer Vision (ICCV). 2015. 121

[88] Glorot, X., Bengio, Y. Understanding the difficulty of training deep feedforward neural networks. tomo 9 de *Proceedings of Machine Learning Research*, págs. 249–256. Chia Laguna Resort, Sardinia, Italy: JMLR Workshop and Conference Proceedings, 2010. URL http://proceedings.mlr.press/v9/glorot10a.html. 121

[89] Wikipedia contributors. Cross entropy — Wikipedia, the free encyclopedia, 2020. URL https://en.wikipedia.org/w/index.php?title=Cross_entropy&oldid=983515385, [Online; accessed 21-October-2020]. 126

[90] Bottou, L. Online algorithms and stochastic approximations. En: D. Saad (ed.) Online Learning and Neural Networks. Cambridge, UK: Cambridge University Press, 1998. URL http://leon.bottou.org/papers/bottou-98x, revised, may 2018. 130

[91] Sutskever, I., Martens, J., Dahl, G., Hinton, G. On the importance of initialization and momentum in deep learning. *30th International Conference on Machine Learning, ICML 2013*, págs. 1139–1147, 01 2013. 132

[92] Nesterov, Y. A method for solving the convex programming problem with convergence rate $o(1/k^2)$. *Proceedings of the USSR Academy of Sciences*, **269**, 543–547, 1983. 132

[93] Nesterov, Y. Introductory lectures on convex optimization - a basic course. En: Applied Optimization. 2004. 132

[94] Duchi, J., Hazan, E., Singer, Y. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, **12** (61), 2121–2159, 2011. URL http://jmlr.org/papers/v12/duchi11a.html. 133

[95] Hinton, G. Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012. 134

[96] Kingma, D. P., Ba, J. Adam: A method for stochastic optimization, 2017. 135

[97] Reddi, S. J., Kale, S., Kumar, S. On the convergence of adam and beyond, 2019. 136

[98] Zeiler, M. D. Adadelta: An adaptive learning rate method, 2012. 136

[99] keras. Keras api reference, 2020. URL https://keras.io/api/. 137

[100] Tensorflow. Tensorflow api documentation, 2020. URL https://www.tensorflow.org/api_docs/. 137

[101] Wikipedia contributors. Precision and recall — Wikipedia, the free encyclopedia, 2020. URL https://en.wikipedia.org/w/index.php?title=Precision_and_recall&oldid=983564675, [Online; accessed 26-October-2020]. 139, 140

[102] Koehrsen, W. Beyond accuracy: Precision and recall. URL https://towardsdatascience.com/beyond-accuracy-precision-and-recall-3da06bea9f6c. 139

[103] Wikipedia contributors. F-score — Wikipedia, the free encyclopedia, 2020. URL https://en.wikipedia.org/w/index.php?title=F-score&oldid=984903554, [Online; accessed 26-October-2020]. 140

[104] Kim, B. K., Kang, H., Park, S. Drone classification using convolutional neural networks with merged doppler images. *IEEE Geoscience and Remote Sensing Letters*, **14** (1), 38–42, 2017. 175

[105] Göksu, H. Ground moving target recognition using log energy entropy of wavelet packets. *Electronics Letters*, **54**, 233–235, 2018. 175

[106] Tompson, J., Goroshin, R., Jain, A., LeCun, Y., Bregler, C. Efficient object localization using convolutional networks, 2015. 176

[107] van den Oord, A., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., *et al.* Wavenet: A generative model for raw audio, 2016. 176